RUNTIME SPECIALIZATION FOR HETEROGENEOUS CPU-GPU PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Naila Farooqui

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2016

RUNTIME SPECIALIZATION FOR HETEROGENEOUS CPU-GPU PLATFORMS

Approved by:

Dr. Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
College of Electrical and Computer Engineering and School of Computer Science
Georgia Institute of Technology

Dr. Vanish Talwar
Research Scientist
PernixData, Inc.

Dr. Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Dr. Rajkishore Barik
Research Scientist
Intel Labs

Date Approved: October 19, 2015

To my children – with the hope to inspire them to never give up

ACKNOWLEDGEMENTS

My Ph.D. journey has been a non-traditional one – with young children, across continents, and through multiple internships that eventually laid the foundations of my thesis work. Such an enormous undertaking cannot be achieved without a strong support system, and I have been exceptionally fortunate to have had some of the strongest, brightest, and kindest people nurture me throughout this journey. I take this opportunity to thank them for all that they have done for me.

One of the most valuable treasures of my Ph.D. experience was my relationship with my advisor, Professor Karsten Schwan. In the last five years, he has nurtured me, inspired me, and motivated me not only to be a great researcher, but more importantly, to be a great human being. At the beginning of almost each year of my study, I put forth a new logistical dilemma in front of him, seeking his support and understanding: from sharing the news of a new baby's arrival to wanting to join my husband in Tokyo for his short-term job assignment, to relocating to the Bay Area prior to completing my degree. His unwavering faith in me, kindness and understanding, and diligence in supporting me through every obstacle that came my way have made getting my Ph.D. one of the most exciting and rewarding experiences of my life. Today, there are no words to express how incomplete it feels to have reached this milestone without him. Twenty-one days before my defense, I lost Professor Karsten Schwan to cancer. All my dreams of having him with me through the final moments of "our" glory, celebrating the end of this beautiful journey, have been shattered. And I was left with just the hopeful image of seeing him there, in that room, watching and cheering for me, as I presented my defense. I miss you, Professor Karsten Schwan, and I hope that you are proud of me.

I'm also greatly indebted to my co-advisor, Professor Sudhakar Yalamanchili, whose insights, wisdom, and support shaped both my research and my perspectives as an individual. Professor

Yalamanchili inspired me to seek that which makes me truly happy, to look inside and follow my heart, and to not be swayed by trivial matters but instead to hold steadfast to the things that truly matter. He taught me to enjoy the journey, for in the end, it is the journey that matters. He also was one of the strongest champions of my work, and always made me realize the significance of my contributions. His positive energy and constant encouragement helped me achieve my true potential as a researcher.

I'd also like to thank my committee members, Dr. Richard Vuduc, Dr. Ada Gavrilovska, Dr. Vanish Talwar and Dr. Rajkishore Barik, for their insightful comments, feedback, and advice. In addition, Dr. Ada Gavrilovska has been a great friend and colleague to just share daily nuances of research life with.

Three out of four of the major contributions of my thesis work were by-products of my tremendously fulfilling internship experiences. I would like to express my heartfelt gratitude to my mentors at Microsoft Research, Dr. Chris Rossbach and Dr. Yuan Yu, at HP Labs, Dr. Vanish Talwar, Dr. Indrajit Roy, and Dr. Yuan Chen, and at Intel Labs, Dr. Tatiana Shpeisman, Dr. Rajkishore Barik and Dr. Brian Lewis, for their guidance, encouragement, and insights. These are some of the brightest researchers I've met, and they not only accepted me as their student but helped me reach the highest standards of research. I am humbled and honored to have had their support and contributions as part of my most significant achievement.

I've also had the privilege to work with some of the most outstanding students at Georgia Tech. Andrew Kerr, a close friend and my first true mentor, walked me through the essentials of writing a high-quality research paper and instilled in me the importance of building my research components as product deliverables rather than mere prototypes. Greg Diamos and Vishakha Gupta have been role-model research students as well as excellent friends, inspiring me to give my best from their examples. Alex Merritt, Haicheng Wu, Jin Wang, and Priyanka Tembey have been great friends and colleagues.

My deepest gratitude goes to my family for their unrelenting love and support throughout my

life. My older brother, Adnan Farooqui, is my role-model and the best teacher I ever had. Adnan taught me to truly cherish my work, to look beyond a good grade or society's standards, and to instead seek inner happiness and satisfaction. For Adnan, material success was never important. He valued success in his relationships, in his tireless and selfless love towards others, and in the integrity of his work. He taught me to pursue my passions with full force, while keeping grounded in my principles. My younger brother, Farhan Farooqui, reminded me to stop and smell the roses along the way. His nonchalant, happy-go-lucky personality has always made life look easy. His presence alone has meant joy and happiness for all those around him.

I'd also like to thank my two cousin-sisters, Lubna Qureishi and Leila Qureishi, for always making me feel like a rock-star in whatever I do! My sisters celebrated every single one of my achievements, no matter how small, throughout this journey. I will always cherish our numerous coffee shop dates that lifted me up in times of stress and multiplied my joys in times of success.

I am immensely grateful to my children, Isa and Alia, for making my Ph.D. journey all the more worthwhile! Even at such a young age, they were completely in-tune with my aspirations and emotions. I am extremely fortunate to have such bright and wonderful children, who always bring so much meaning and value to everything I do. At the same time, I worked endlessly to give them my 100%, which meant that their time had to be entirely theirs. So no matter what I was working on, I had to stop myself when they needed me. And while this may seem counter-productive, my children were responsible for many of my research break-throughs, for it was often those unintended breaks that enabled me to look at a problem with a refreshed perspective, and make progress in ways that I had not imagined.

At the crux of my support system is my husband, Najam Baig. Najam defines the very essence of companionship and togetherness in my life. To say that he supported me through my trials and tribulations as a Ph.D. student is an under-statement. In reality, Najam has lived through all of them as if they were his own. He has not only been a wonderful life-partner, but an exceptional father as well. One of my most precious memories is when I took my Ph.D. qualifier exam from

Tokyo, Japan. Even though I was in Tokyo, I was required to take the exam at the same time as the other students, who were taking it from Atlanta. This meant that while others were scheduled to take it from 8 am to 5 pm Eastern time, I had to take it from 10 pm to 6 am Japan time! At that time, our daughter, Alia, was only four months old and could not sleep through the night without me. I remember Najam struggling through that entire night, trying to get her to calm down and sleep. At about 3 a.m., an extremely exhausted Najam came into the room with Alia in his arms, and said to me: "Look Love, I did it! She's finally sleeping!" He slowly put her down on the bed, and collapsed on the side, with his arms still underneath her. That picture has been captured in my memory forever. And in fact, many such memories have defined my Ph.D. journey. Thank you, Najam, for always being there for me.

Finally, I would like to thank the two people who undoubtedly have made the greatest contribution in not just my Ph.D. but in every single aspect of my life: my father, Dr. Mohammad Yahya Farooqui, and my mother, Dr. Zakia Jabbar Farooqui. I would not have reached this point in my life if it weren't for my dad's faith in me. My brothers and I were raised in Riyadh, Saudi Arabia, and my parents sent all of their children to the United States to pursue higher education. My older brother was living in the United States, having just finished his undergraduate studies, when it was my turn to start college. At a time when everyone questioned my father's decision to send his 17-year old daughter alone to the United States, my father didn't even blink an eye. It was simple and clear to him: he had made sure his son had the opportunity to study abroad, and he was going to make sure his daughter got the same privilege. In fact, my father has always had the most faith in my abilities. As far as he was concerned, his daughter could never fail. My father is my greatest pillar of strength. Whenever I close my eyes in moments of weakness or fear, I always see my father next to me. It is from him that I derive my inner strength; to work hard, to never give up, and to create my own opportunities.

My mother is my idea of perfection. Her patience, resilience, kind-heartedness, and intellectual ability are unparalleled. She is my best friend and confidante, my soul-mate and companion

through every phase of my life. I have shared every single aspect of my journey with her, from the details of each paper review, to what I am going to wear at a given venue, to what career path to choose for myself. Many have asked me how I've managed to make progress on my work with all the responsibilities of parenting and motherhood. Well, it's because I have what many don't have: the most amazing mother in this world. During my pregnancy and daughter's birth, my husband had to stay abroad for months at a stretch, and it was my mother who took care of all of us - me, my two year old son, and my unborn daughter. She lived with us, managed our home, made the most scrumptious meals, and supported me through all the stresses and difficulties of pregnancy, childbirth, and parenting. Without my mother, nothing in life would have been possible for me.

My parents have shown me the value of hard work, resilience to failure, and perseverance. At the end, the Ph.D. is really nothing more than a test of one's perseverance and resilience. The greatest joy from this achievement comes from knowing that I have followed my parents' example, and I hope this achievement becomes an example for my children as well, to inspire them to never give up, in the same way that my parents have inspired me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Heterogeneous parallel architectures like those comprised of CPUs and GPUs are a tantalizing compute fabric for performance-hungry developers. While these platforms enable order-of-magnitude performance increases for many data-parallel application domains, there remain several open challenges: (i) the distinct execution models inherent in the heterogeneous devices present on such platforms drives the need to dynamically match workload characteristics to the underlying resources, (ii) the complex architecture and programming models of such systems require substantial application knowledge and effort-intensive program tuning to achieve high performance, and (iii) as such platforms become prevalent, there is a need to extend their utility from running known regular data-parallel applications to the broader set of input-dependent, irregular applications common in enterprise settings.

The key contribution of our research is to enable runtime specialization on such hybrid CPU-GPU platforms by matching application characteristics to the underlying heterogeneous resources for both regular and irregular workloads. Our approach enables profile-driven resource management and optimizations for such platforms, providing high application performance and system throughput. Towards this end, this research: (a) enables dynamic instrumentation for GPU-based parallel architectures, specifically targeting the complex Single-Instruction Multiple-Data (SIMD) execution model, to gain real-time introspection into application behavior; (b) leverages such dynamic performance data to support novel online resource management methods that improve application performance and system throughput, particularly for irregular, input-dependent applications; (c) automates some of the programmer effort required to exercise specialized architectural features of such platforms via instrumentation-driven dynamic code optimizations; and (d) proposes a specialized, affinity-aware work-stealing scheduling runtime for integrated CPU-GPU processors that efficiently distributes work across all CPU and GPU cores for improved load balance, taking into account both application characteristics and architectural differences of the underlying devices.

CHAPTER I

INTRODUCTION

There are two important trends in computing today: (i) the pervasiveness of heterogeneous parallel architectures, such as general-purpose GPUs coupled with CPUs, and (ii) the importance of rapid, predictive data analytics workloads in a variety of industry sectors. Parallel hardware, such as GPUs, can deliver order of magnitude performance increases to high-performance applications, since GPU hardware is designed explicitly to take advantage of regularity characterized in workloads that exhibit minimal synchronization, high arithmetic intensity, and predictable memory access and control-flow patterns. Given the ubiquity of GPUs in modern computing environments, performance-hungry programmers also use GPUs in increasingly complex applications where algorithms rely fundamentally on unstructured or irregular control and data access patterns. Many predictive data analytics workloads, in particular graph algorithms, exhibit such irregularity. This has given rise to extensive work on efficiently running such applications on today's heterogeneous CPU-GPU systems – new techniques for algorithm parallelization and graph partition [58,61], specialized machine learning and graph frameworks [71], and runtime support to accelerate analysis using GPU platforms [16, 88, 102].

1.1 A Case for Runtime Specialization

While heterogeneous GPU platforms can improve performance over traditional multi-core CPU platforms for a variety of data analytics workloads [23, 67, 95, 97], there remain several open challenges. First, the distinct execution models inherent in the heterogeneous devices present on such platforms drives the need to dynamically match workload characteristics to the underlying resources. Figure 1 shows the kernel runtime speedups of a variety of predictive data analytics

workloads on the GPU compared to multi-core CPU execution. As shown in the plot, GPUs benefit compute-intensive applications that feature minimal synchronization, uniform control flow and regular memory access patterns, such as Black-Scholes (BS) and K-Means (KM). Applications exhibiting larger regions of serialized code or irregular control-flow are better suited for the CPU's execution model. Irregular applications, such as Page-Rank (PR), can be 3× faster on CPUs, based on its input. Page-Rank is a link analysis algorithm used to rank websites in search engine results. Its computation is highly dependent on the structure of its input. Therefore, the benefit an application derives from a device depends on runtime characteristics, such as control-flow irregularity and memory bandwidth requirements.

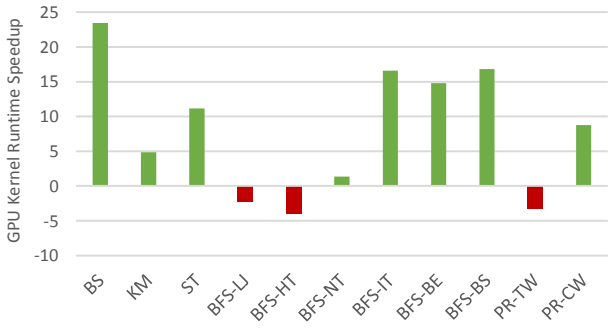

Figure 1: Speedups of GPU execution over multi-core CPU execution for a variety of predictive data analytics workloads, running on AMD A10-5800K APU desktop, equipped with Radeon HD 7660D GPU.

Second, the complex architecture and programming models of heterogeneous GPU systems require substantial application knowledge and effort-intensive program tuning to achieve high performance. To demonstrate the significance of applying application-specific knowledge in designing efficient GPU algorithms, we use Breadth-First-Search as an example. Breadth-First-Search (BFS) is a fundamental building block for many graph-based applications, including connected components detection, shortest path algorithms and betweenness centrality. There are two existing state-of-the-art GPU implementations for BFS, which we refer to as ALG1 [64] and ALG2 [43]. ALG1 uses a hierarchical queue to reduce overheads associated with a single, global task queue for

the entire GPU. ALG2 uses a warp-centric programming method to address workload imbalance, characterized as thread divergence in GPU programming. While ALG1 provides a more general optimization for the GPU, ALG2 specifically improves the performance of irregular graphs, defined as exhibiting a large skew in degree distribution, and therefore a more significant workload imbalance. Figure 2 presents the speedup of the two algorithms on the GPU over multi-core CPU execution for multiple input graphs, both regular and irregular graphs. The plot shows that for regular graphs, ALG1 performs substantially better than ALG2, while for irregular graphs, ALG2 can be 3× faster than ALG1.

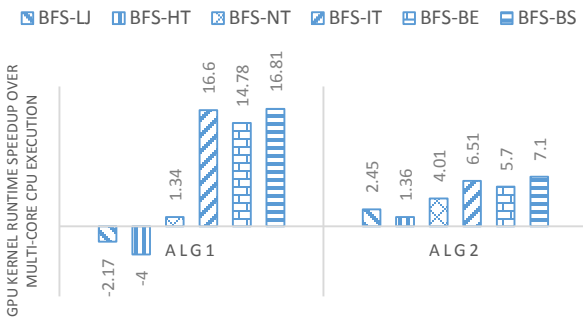

Figure 2: Speedups of GPU execution over multi-core CPU for the two BFS algorithms on various BFS input graphs, running on AMD A10-5800K APU desktop, equipped with Radeon HD 7660D GPU. The first three bars represent *irregular* graphs, while the latter three represent *regular* graphs.

The BFS results not only highlight the need for using application-specific knowledge to achieve high performance, but also demonstrate the input-dependent performance of such irregular applications. More generally, a third challenge with achieving high performance on heterogeneous GPU platforms is that the performance of irregular applications with input-dependent and time-varying runtime behaviors is difficult to predict statically. We motivate the need for dynamic optimization methods using the *SkyServer* [103] application. *SkyServer* takes as input large collections of astronomical, digital data in the form of photo objects and neighbors, and filters them to find related items. The SkyServer workload is, in essence, a series of relational equi-join operations and filtering over the two collections. Differing input data distributions can yield very different

selectivity for the join predicates, which in turn has a profound impact on dynamic memory access and control flow patterns in the GPU code implementing the join. Figure 3 shows the performance for two distinct input sets, *Input Set 1*, where the data distribution yields very low selectivity for the join predicate, and *Input Set 2*, where the majority of neighbor objects match the join predicate. Specifically, the plot compares the performance of two distinct data layouts for the given input sets. A direct mapping generates an Array-of-Structures (AoS) data layout in GPU memory, which prevents coalesced reads and writes as the members of the data structure are placed contiguously in memory, forcing different threads to access scattered memory locations. The optimized Structure-of-Arrays (SoA) layout, which results in a sequential access pattern for all threads in the same warp, improves memory coalescing by a factor of two. However, as shown in Figure 3, the SoA version improves performance for the first input set but has a negative impact on the second input set. This result highlights the need to take into account dynamic information about the application to deal with input-dependent performance.

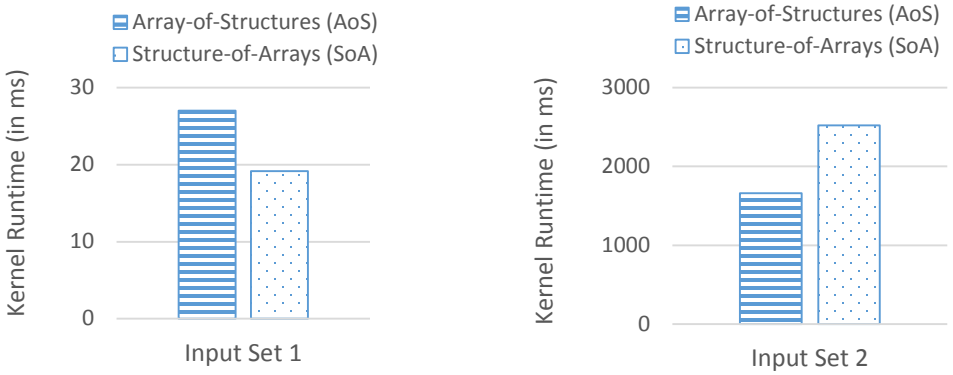

Figure 3: Kernel runtimes for the AoS and SoA optimizations on two distinct input sets. Experiments conducted on NVIDIA Tesla M2075 GPU.

In summary, all of these challenges make a case for **runtime specialization**. In this context, runtime specialization is defined as using *online* methods to match *dynamic* workload characteristics to underlying resources, in order to drive greater efficiency. Our research addresses precisely

this need for heterogeneous CPU-GPU platforms.

1.2 Thesis Statement

In this dissertation we posit that, for today's diverse set of data-intensive applications, it is important to leverage real-time introspection into application behavior to drive resource management and optimization methods on heterogeneous platforms. *Runtime specialization can enable performance scaling of data-intensive applications across emerging heterogeneous CPU-GPU platforms.*

Our approach leverages dynamic instrumentation and online profiling methods for real-time application introspection, as well as novel scheduling techniques, which take into consideration the underlying architectural differences of such heterogeneous platforms. We believe that runtime specialization can provide significant efficiency gains, for both regular and irregular applications on heterogeneous machines, without the need for developers to explicitly profile and manually tune their codes.

1.3 Solution Approach

The objective of this dissertation is to demonstrate that runtime specialization can achieve high performance and platform throughput without the need for manual intervention for application profiling and/or tuning, for diverse data-intensive applications on heterogeneous CPU-GPU platforms. The first contribution of our research, therefore, is a dynamic instrumentation framework that provides real-time insights into application behavior seamlessly, transparently, and efficiently for GPU-based platforms. We then use the online instrumentation and profiling methods to (a) drive better resource management, and (b) perform profile-guided optimizations to improve both platform throughput and individual application performance. Finally, as a proof of our hypothesis, we use runtime specialization to improve a classical, well-known resource management paradigm, work-stealing. Each of these contributions manifest themselves into different constituents of our research, as highlighted below:

- **Lynx** [35] is a dynamic instrumentation engine for data-parallel applications on GPU-based architectures. Lynx provides the necessary real-time introspection capabilities into an application's runtime behavior to support *dynamic, online* methods for resource management and optimizations. Specifically, it provides an extensible set of C-based language constructs to build customizable program analysis tools that target the data-parallel programming paradigm used in GPUs. Furthermore, it uses a just-in-time (JIT) compiler to translate, insert and optimize instrumentation code at the intermediate representation (IR) layer. In an nutshell, Lynx provides the capability to write instrumentation routines that are (1) *selective*, instrumenting only what is needed, (2) *transparent*, without changes to the applications' source code, (3) *customizable*, and (4) *efficient*.

- **Luminar** is a runtime framework that leverages Lynx to enable online resource management on integrated CPU-GPU platforms. In particular, Luminar combines the advantages of integrated GPU systems, such as reduced overhead of offloading computation to the GPU and potential for fine-grained concurrent resource scheduling, with real-time introspection into application behavior provided by Lynx to deliver significant gains in application performance, system throughput, and energy efficiency when co-scheduling a mix of irregular and regular workloads.

- **Leo** [37] is a profile-driven, dynamic optimization framework for GPU applications, which also leverages Lynx to drive GPU-specific code optimizations. While GPUs enable order-of-magnitude performance increases in many data-parallel application domains, writing efficient codes that can actually manifest those increases is a non-trivial endeavor, typically requiring developers to exercise specialized architectural features exposed directly in the programming model. Achieving good performance on GPUs involves effort-intensive tuning. Leo aims to automate much of this effort using dynamic instrumentation to inform dynamic, profile-driven optimizations.

- **Libra** is an affinity-aware work-stealing scheduler for integrated CPU-GPU processors that efficiently distributes work at runtime across all CPU and GPU cores for improved load balance. Libra differs from classical work-stealing by including techniques to deal with the architectural differences between the CPU and GPU in an integrated processor, such as different clock frequencies, costs for atomic operations, and latencies in accessing caches and shared memory. Such architectural differences hurt the performance that can be obtained using classical work-stealing. In addition, Libra effectively supports both regular and irregular applications.

The remainder of the dissertation is organized as follows. Chapter 2 provides a brief overview on GPU computing and application domains used in this work. Chapter 3 describes Lynx, the dynamic instrumentation engine that provides real-time introspection into application behaviors for GPU-based platforms. Chapter 4 presents the Luminar runtime framework, which uses Lynx to enable dynamic resource management methods for integrated CPU-GPU systems. Chapter 5 describes the Leo optimization framework. Leo shows how Lynx can be leveraged to drive profile-driven code optimizations to improve GPU-specific codes. Chapter 6 presents Libra, an affinity-aware, work-stealing scheduler that improves upon classical work-stealing for integrated CPU-GPU platforms by incorporating workload characterization and CPU-GPU architectural differences. Chapter 7 summarizes the finding of this research and comments on future investigations and applications of instrumentation-driven runtime specialization.

CHAPTER II

BACKGROUND

The primary objective of this dissertation is to demonstrate the effectiveness of runtime specialization in improving performance of both regular and irregular applications on heterogeneous CPU-GPU platforms. The techniques we use exploit both the heterogeneity in the execution models of the underlying devices, and diversity in applications' runtime behaviors. It is therefore important to understand different facets of GPU computing, including the GPU execution model and its programming frameworks, as well as the application domains our system targets to highlight runtime specialization.

2.1 GPU Execution Model

A data-parallel program is composed of a series of multi-threaded *kernels*, which can be thought of as computational functions that are executed on the GPU. Computations are performed by a tiered hierarchy of threads. At the lowest level, collections of threads are mapped to a single *SIMD (Single-Instruction Multiple-Data)* core and executed concurrently. Each SIMD core includes an L1 data cache, a shared scratch-pad memory for exchanging data between threads, and a SIMD array of functional units. This collection of threads is known as a *thread block*, and kernels are typically launched with tens or hundreds of thread blocks which are oversubscribed to the set of available SIMD cores.

A work scheduler on the GPU maps thread blocks onto individual SIMD cores for execution, and the programming model forbids global synchronization between SIMD cores except on kernel boundaries. Global memory is used to buffer data between kernels as well as to communicate between the CPU and GPU. While shared memory is on-chip, global memory is off-chip and

therefore has the highest latency in the memory hierarchy.

Thread blocks execute in SIMD chunks called *warps* in NVIDIA terminology, and *wavefronts* in OpenCL terminology. A *warp*, or a *wavefront*, is the most basic unit of scheduling on the GPU, and is defined as the maximal subset of threads within a thread block which processes a single instruction over all of the threads in it at the same time, in lock-step fashion. Hardware warp/wavefront and thread scheduling hide memory and pipeline latencies. When threads within a warp diverge due to data-dependent conditional branches, the GPU will instruct the warp to execute one branch first, deactivating all threads that evaluated to false for that branch, and then proceed to the other branch, reversing the situation. As a result, thread divergence within a warp leads to serialized execution, which can result in a significant performance loss.

2.2 GPU Programming Frameworks

CUDA [74] toolchain is a programming language and API that enables data-parallel programs to be written in a language with C++-like semantics. Open Computing Language (OpenCL) [96] is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and a wide array of other accelerators/processors, including DSPs and FPGAs. OpenCL specifies a language (based on C99) for programming these devices and application programming interfaces (APIs) to control the platform and execute programs on the compute devices. Both CUDA and OpenCL are popular programming frameworks for GPU computing. However, CUDA was developed by NVIDIA and is only supported for NVIDIA GPUs, while OpenCL has been adopted by multiple GPU vendors, including AMD, Intel, and NVIDIA. Additionally, since the same OpenCL can be executed on different processors, such as either the CPU or GPU, it is an ideal choice for scenarios that support the ”write-once-run-anywhere (WORA)” programming model.

2.3 Application Domains

To highlight the need for runtime specialization, we focus on applications that exhibit highly input-dependent, time-varying runtime behaviors, such as graph applications. The performance of such applications is difficult to predict statically and therefore they are prime candidates for runtime specialization. We also include applications that exhibit uniform control-flow, regular memory-access patterns with minimal synchronization, and therefore cater to GPU's execution model, such as `Black-Scholes`, `Matrix-Multiply`, and `K-Means`, as well as applications with larger regions of serialized code that are better-suited to the CPU, such as `Face-Detection`, `Substring Finder`, and `Barnes-Hut`, in order to demonstrate that our runtime frameworks provide high performance to a diverse set of applications. Many of our workloads come from NVIDIA CUDA SDK [74], AMD SDK [6], Parboil [44] and Rodinia [28] application suites. We have also implemented existing state-of-the-art algorithms for graph applications, such as `BFS` [43, 64] and `Page-Rank`.

CHAPTER III

LYNX: A DYNAMIC INSTRUMENTATION SYSTEM FOR DATA-PARALLEL APPLICATIONS ON GPU PLATFORMS

3.1 Introduction

Instrumentation is a technique of inserting additional procedures into an application to observe or improve its behavior. Dynamic binary instrumentation involves inserting code at the instruction level of an application while the application is executing. Such instrumentation offers opportunities for detailed low-level inspection such as register spilling, instruction layout, and code generated downstream from compiler optimization passes. It also offers the capability to attach/detach instrumentation at runtime, thus selectively incurring potential overheads only when and where instrumentation is required.

Although dynamic instrumentation has been proven to be a useful program analysis technique for traditional architectures [68], it has not been fully exploited for GPU-based systems. This is in part due to the vendor-specific instruction set architectures (ISAs) and limited tool-chain support for direct modification of native binaries of GPUs.

Lynx is a dynamic binary instrumentation infrastructure for constructing customizable program analysis tools for GPU-based, parallel architectures. It provides an extensible set of C-based language constructs to build program analysis tools that target the data-parallel programming paradigm used in GPUs. The highlights of Lynx's design, therefore, are *transparency, selectivity, customization, and efficiency*.

Transparency is achieved by inserting instrumentation at the Parallel Thread eXecution (PTX) level [73], ensuring that the original CUDA applications remain unchanged. PTX is the virtual

instruction set targeted by NVIDIA's CUDA [74] and OpenCL [96] compilers, which implements the Single-Instruction Multiple-Thread (SIMT) execution model. Furthermore, the instrumentation generated with Lynx does not alter the original behavior of the application.

Selective instrumentation implies specification of the instrumentation that is needed and where it should be inserted. With Lynx, instrumentation can be inserted at the beginning or end of kernels and basic blocks, or at the instruction level. It is also possible to instrument only particular classes of instructions.

Lynx makes such **customization** of instrumentation easy to do because the instrumentation for data-parallel programs is specified and written with a subset of the C programming language, termed C-on-Demand (COD). Further, the Lynx instrumentation APIs capture major constructs of the data-parallel programming model used by GPUs, such as threads, thread blocks, and grids. This results in a low learning curve for developers, particularly those already familiar with CUDA or OpenCL.

Lynx provides **efficient** instrumentation by using a JIT compiler to translate, insert and optimize instrumentation code. Additionally, the Lynx API supports the creation of instrumentation routines specifically optimized for GPUs. An example is warp-level instrumentation, which captures the behavior of a group of threads as opposed to individual threads. By taking advantage of the lock-step execution of threads within a warp, it becomes possible to reduce the memory bandwidth costs incurred by instrumentation.

Lynx also provides portability by enabling support across several processor back-ends, including various GPU vendors (e.g. NVIDIA, AMD, Intel) as well as across discrete and integrated GPU platforms. Lynx's highly modular design makes it amicable to extending support to different intermediate representations (IRs) and GPU runtimes (e.g. OpenCL and CUDA). In its current implementation, Lynx includes runtime support for both OpenCL and CUDA, and instrumentation support for NVIDIA's Parallel Thread eXecution (PTX) [73], AMD's Intermediate Language (IL) [8], and LLVM [55] intermediate representations (IRs).

In summary, the technical contributions of Lynx are as follows:

- A dynamic instrumentation system for GPGPU-based, parallel execution platforms that supports transparency, selectivity, customization, and efficiency.
- Implementation and design of a C-based language specification for defining instrumentation for data-parallel programs, which supports the construction of custom instrumentation routines.
- Demonstration of the versatility and usefulness of the Lynx instrumentation system with implementations of performance metrics applicable to GPGPU-based architectures.
- Evaluation of Lynx's performance and instrumentation overheads for a variety of GPU workloads, specifically highlighting optimization opportunities on GPUs.

The experiments in this chapter were performed on a system with an Intel Core i7 running Ubuntu 10.04 x86-64, equipped with an NVIDIA GeForce GTX 480, except for the experiment comparing Lynx's performance with Ocelot's emulator, which was performed on an Intel Xeon X5660 CPU machine equipped with an NVIDIA Tesla C2050, and the evaluation in Section 3.5.3, which was done on an AMD Radeon HD 7770 GPU. Benchmark applications for experiments are chosen from the NVIDIA CUDA SDK [74], the Parboil Benchmark Suite [44], and the Rodinia Benchmark Suite [28].

3.2 Design and Implementation

In this section, we present a system overview of Lynx, followed with a detailed discussion on Lynx's run-time flow, using NVIDIA's CUDA and PTX IR as driving examples. Note, however, that the same concepts apply to instrumenting OpenCL applications running on either AMD GPUs (with AMD IL) or Intel GPUs (with LLVM IR).

3.2.1 System Overview

The Lynx instrumentation system provides the following new capabilities:

- A C-based language specification for constructing customized instrumentation routines
- A C-based JIT compilation framework, which translates instrumentation routines to a given IR
- An instrumentation pass to modify existing kernels with generated instrumentation code
- Run-time management of instrumentation-related data structures

The Lynx system is comprised of an *Instrumentation API*, an *Instrumentor*, a *C-on-Demand (COD) JIT Compiler*, a *C-to-IR Translator*, and a *C-to-IR Instrumentation Pass*. The system's implementation provides the following components: implementations of both *CUDA and OpenCL Runtimes*, *IR Parsers*, *IR-IR Transformation Pass Managers*, and *IR Translators*, with support for multiple backed targets.

We now describe how a CUDA application is instrumented with GPU Lynx. When a CUDA application is linked with GPU Lynx, API calls to CUDA pass through Lynx's CUDA Runtime, which provides a layer of compilation support, resource management, and execution. The CUDA application is parsed into PTX modules, consisting of one or more PTX kernels. If the application is being instrumented, a C-based instrumentation specification is provided to the framework. The Instrumentor serves as the run-time engine for generating the instrumented PTX kernel from the C specification and the original PTX kernel, by enlisting the COD JIT Compiler, the C-to-PTX Translator, and the PTX-PTX Transformation Pass Manager. The COD JIT Compiler produces a RISC-based IR for the C specification and the C-to-PTX Translator generates equivalent PTX instructions. The specification defines where and what to instrument, using the instrumentation target specifiers discussed in the next section. The C-to-PTX Instrumentation Pass, invoked by the

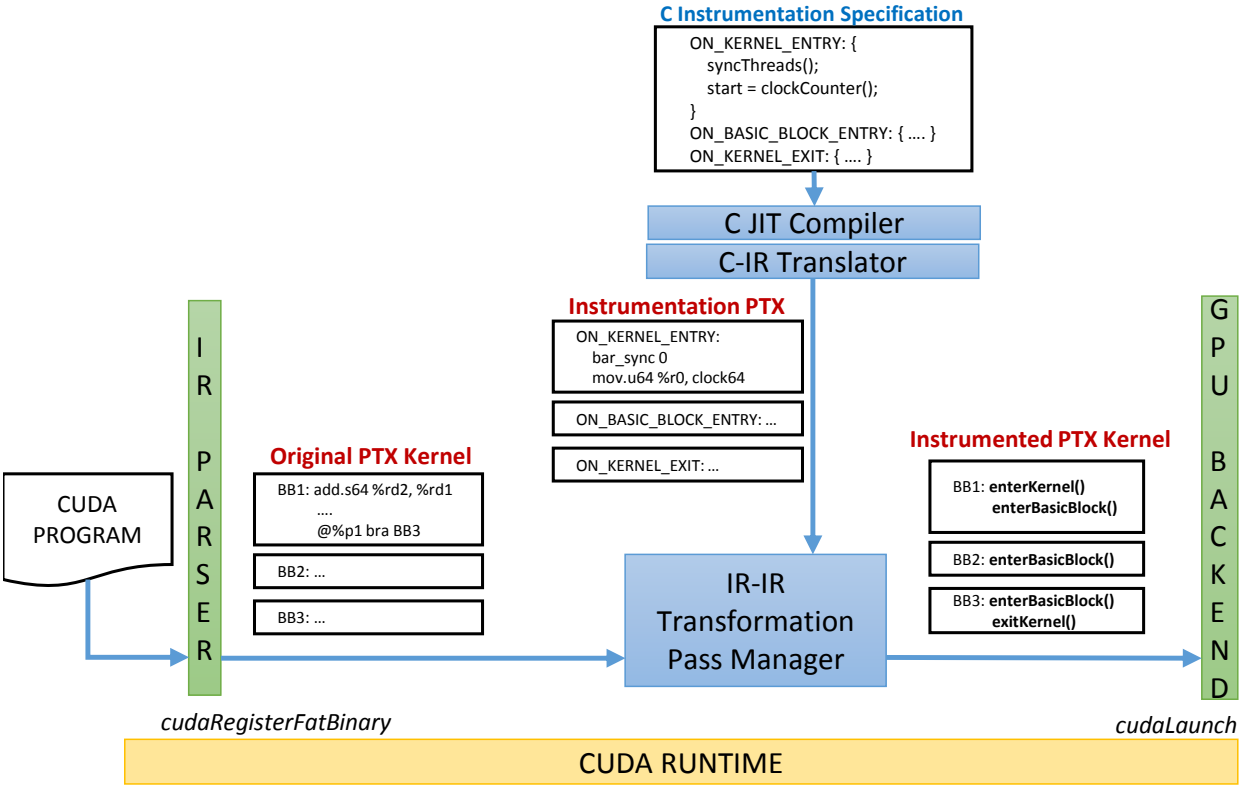

Figure 4: Lynx's run-time/execution flow, depicted with NVIDIA's CUDA Runtime API and PTX IR

Pass Manager, inspects these specifiers and inserts the translated instrumentation PTX accordingly into the original PTX kernel(s).

3.2.2 Execution Flow

Figure 4 illustrates Lynx's execution/run-time flow in the context of CUDA applications. CUDA applications compiled by `nvcc` are converted into C++ programs, with PTX kernels embedded as string literals. When such a program links with our framework, the CUDA Runtime API function, `cudaRegisterFatBinary`, parses these PTX kernels into an internal representation. The original PTX kernel is provided as input to GPU Ocelot's Pass Manager, together with the instrumentation PTX generated from the C code specification via the COD JIT Compiler and the C-to-PTX Translator. The Pass Manager applies a sequence of PTX kernel transformations to the original PTX kernel. A detailed discussion of the Pass Manager and PTX transformation passes can be found in our earlier work [36].

A specific pass, C-to-PTX Instrumentation Pass, is implemented as part of the Lynx framework to insert the generated PTX into the original PTX kernel, according to Lynx's language specification. The final output, the instrumented kernel, is prepared for native execution on the selected device by the PTX Translator/Code Generator.

Since GPU Lynx implements the CUDA Runtime API as well, it enables the insertion of hooks into the runtime system for managing resources and data structures needed to support instrumentation. The Lynx framework utilizes this capability via the Instrumentor component. Its general approach for managing instrumentation-related data for discrete GPUs is to allocate memory on the device, populate the instrumentation-related data structures during kernel execution, and then move the data back to the host, freeing up allocated resources on the device. For integrated GPUs, memory is allocated across shared CPU-GPU buffers, eliminating the need for costly transfers for instrumentation-related data structures.

Table 1: A subset of the Lynx API

Function Name	Description
globalThreadId	Global thread identifier for the current thread.
blockThreadId	Thread identifier for the current thread within its thread block.
blockId	Thread block identifier for the current thread block within its grid.
blockDim	Number of threads per thread block.
syncThreads	Barrier synchronization within a thread block.
leastActiveThreadInWarp	Determines the least active thread in the current warp.
uniqueElementCount	Total number of unique elements in a given buffer of warp or half-warp size.
basicBlockId	Returns the index of the current basic block.
basicBlockExecutedInstructionCount	Total number of executed instructions in the current basic block.

3.3 Instrumentation with Lynx

The Lynx instrumentation API provides both relevant functions to perform instrumentation in a data-parallel programming paradigm and *instrumentation specifiers*, which designate where the instrumentation needs to be inserted. We first present an overview of the Lynx Instrumentation API, followed by a description of three example instrumentation specifications. We end this section with a discussion on the C-to-IR translator's role in generating IR from the instrumentation specifications.

3.3.1 Instrumentation API

Data-parallel kernels are executed by a tiered hierarchy of threads, where a collection of threads, also known as a thread block, is executed concurrently on a single *stream multiprocessor*, or SM, in NVIDIA terminology. Threads within a thread block execute in SIMT manner in groups called warps. Therefore, the Lynx API at a minimum must capture the notions of thread, thread blocks, warps, and SMs. Table 1 describes a subset of the Lynx API. The table also includes functions such as *basicBlockExecutedInstructionCount()*, which represent attributes that are obtained via static analysis of data-parallel kernels.

The instrumentation specifiers are defined as C labels in the language. They can be categorized into four types: instrumentation targets, instruction classes, address spaces and data types. Table 2 captures all of the currently available specifiers in the Lynx instrumentation language.

Table 2: Available specifiers in the Lynx API

Specifier Type	Available Specifiers
Instrumentation Target	ON_KERNEL_ENTRY, ON_KERNEL_EXIT, ON_BASIC_BLOCK_ENTRY, ON_BASIC_BLOCK_EXIT, ON_INSTRUCTION
Instruction Class	MEM_READ, MEM_WRITE, CALL, BRANCH, BARRIER, ATOMIC, ARITHMETIC
Address Space	GLOBAL, LOCAL, SHARED, CONST, PARAM, TEXTURE
Data Types	INTEGER, FLOATING_POINT

Instrumentation target specifiers have a dual purpose. First, they describe where the instrumentation needs to be inserted. For instance, instrumentation can be inserted at the beginning or end of kernels, beginning or end of basic blocks, or on every instruction. Instrumentation is inserted just before the last control-flow statement when inserting at the end of basic blocks or kernels. Second, the instrumentation target specifiers serve as loop constructs. In other words, the instrumentation routine following the target specifier is applied to *each and every* kernel, basic block, or instruction, depending on the target specifier. Note that if no instrumentation target is specified, the instrumentation is inserted at the beginning of the kernel by default.

A user may only want to instrument certain classes of instructions, or only instrument memory operations for certain address spaces. For this reason, the Lynx API includes instrumentation class and address space specifiers. The multiple address space notion is part of the memory hierarchy model of data-parallel architectures. Threads may access data from various memory spaces, including on-device and off-device memory units. Finally, data type specifiers are provided to categorize arithmetic operations by integer and floating-point functional units.

3.3.2 Example Instrumentation Specifications

Figure 5 shows three example instrumentation specifications defined using Lynx's instrumentation language: a basic-block level instrumentation, an instruction level instrumentation and a kernel level instrumentation. The instrumentation target specifiers are noted in each of the examples.

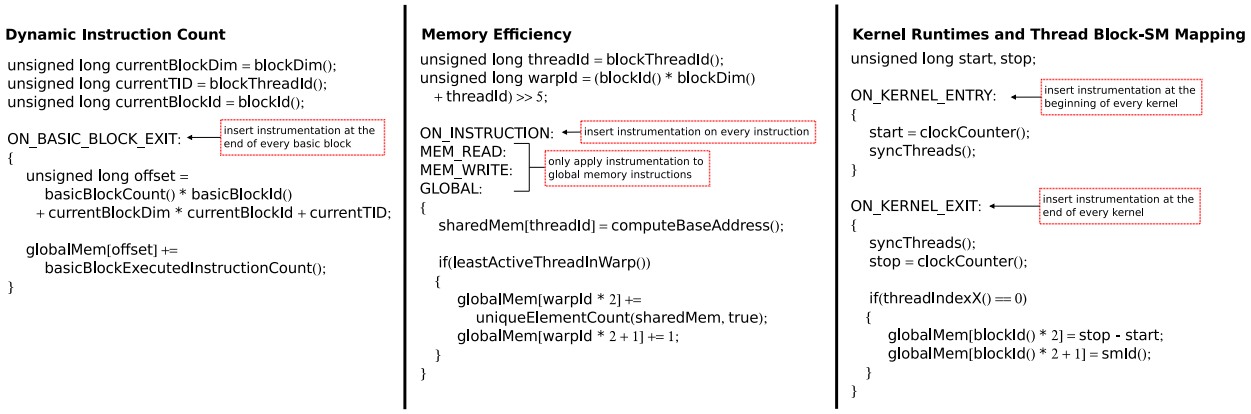

Figure 5: Example Instrumentation Specifications

Dynamic instruction count is a **basic-block** level instrumentation that computes the total number of instructions executed, omitting predicated instructions that do not get executed. The instrumentation results are stored in the global address space of the device in a dynamically allocated array, on a per-basic-block, per-thread-block, per-thread basis. The *globalMem* variable is globally available to store instrumentation data. The actual allocation of this array takes place in the specific instrumentor class that is defined for this metric.

The memory efficiency metric is an example of an **instruction** level instrumentation. For every global load or store instruction, each thread within a thread block computes the base memory address and stores it in shared memory via the *sharedMem* global variable. For NVIDIA GPUs, a half-warp of 16 threads can coordinate global memory accesses into a single transaction. The least active thread in a warp maintains a count of unique memory addresses for each half-warp, to determine the total number of memory transactions required to satisfy a particular global memory request, and increments the dynamic warp count.

Finally, the kernel runtime specification is a **kernel** level instrumentation that uses barrier synchronization for threads, `syncThreads`, within a thread block and the clock cycle counter API, `clockCounter`, to obtain the kernel run-time in clock cycles for each thread block. Since each thread block gets executed on a single SM, the current SM identifier for each thread block, obtained

via `smId`, is also stored in global memory.

3.3.3 C-to-IR Translator

The C-to-IR translator is responsible for parsing the C code specification and translating the RISC-based C IR to the GPU-based IR. Toward this end, the C-to-IR translator walks through each C IR instruction and emits equivalent GPU IR instructions. Special handling occurs for each of the instrumentation specifiers. Since the specifiers are C-style labels, each label is converted to a basic-block label as a place-holder in the generated GPU IR. All of the GPU IR instructions that belong to the specifier become a part of a new translation basic-block. The C-to-IR instrumentation pass then looks for these labels and inserts the corresponding instructions of that translation basic-block into the designated location of the application IR kernel.

Although multiple instrumentation specifiers are allowed, each resulting in its own translation basic-block, the current implementation does not prevent conflicts between the instrumentation of different translation blocks. A possible solution is to enforce that each instrumentation is applied individually to the application's IR kernel as a separate instrumentation pass, effectively resulting in a chain of instrumentation passes executed independently. We have left this implementation as a future enhancement to our framework.

3.4 GPU-Specific Instrumentation

In this section, we discuss GPU-specific instrumentation and present common metrics for GPU-based systems. We also present multiple methods for implementing the dynamic instruction count metric.

3.4.1 Warp-Level Instrumentation

A warp-level instrumentation is one that focuses on the behavior of a *warp*, the collection of threads physically issuing instructions on an SM during the same cycle, rather than the independent

behaviors of each thread. Such instrumentation is sensitive to the way SIMD processors are utilized as well as how memory requests are coalesced before issuing to DRAM. GPUs implement thread divergence through implicit hardware predication; a bit mask stores the set of active threads within a particular warp. Warp-level instrumentation typically involves a reduction query across particular values computed by each thread in a warp. These could include conditional branch predicates, memory addresses, or function call targets.

3.4.2 Performance Metrics for GPUs

3.4.2.1 Memory Efficiency

Memory efficiency is an example of a GPU-specific **warp-level** metric that characterizes the spatial locality of memory operations to global memory. Global memory is the largest block of memory in the PTX memory hierarchy and also has the highest latency. To alleviate this latency cost, the PTX memory model enables coalescing of global memory accesses for threads of a half-warp into one or two transactions, depending on the width of the address bus. However, scatter operations, in which threads in a half-warp access memory that is not sequentially aligned, result in a separate transaction for each element requested, greatly reducing memory bandwidth utilization. The goal of the memory efficiency metric, therefore, is to characterize memory bandwidth utilization by determining the ratio of dynamic warps executing each global memory dynamic instruction to the number of memory transactions needed to complete these instructions. Memory efficiency for several applications, obtained via Lynx's instrumentation system, are presented in Figure 6.

The Lynx instrumentation system provides useful APIs to enable the creation of intricate warp-level instrumentations. For example, for the memory efficiency metric, the base address of each global memory operation is computed and stored for all threads in a thread block. If the base address is the same for all threads belonging to a half-warp, then the memory accesses will be coalesced. A single thread within a warp, determined by the `leastActiveThreadInWarp` API, performs an online reduction of the base addresses written to a shared buffer by all threads

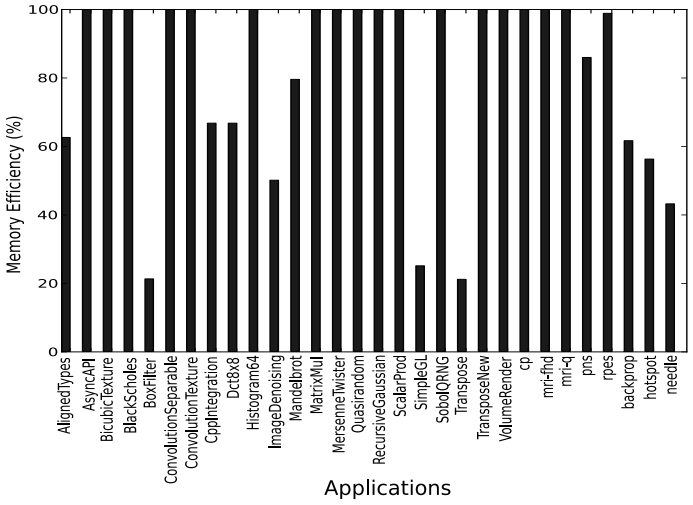

Figure 6: Memory Efficiency

in the warp. The `uniqueElementCount` API, which is used to keep a count of unique base addresses within a half-warp, determines the total number of memory transactions required for a particular memory operation.

3.4.2.2 Branch Divergence

Branch divergence is another example of a warp-level metric. It provides insight into a fundamental aspect of GPU performance, namely its SIMT execution model. In this model, all threads within a warp execute each instruction in lock-step fashion until they reach a data-dependent conditional branch. If such a condition causes the threads to diverge, the warp serially executes each branch path taken, disabling threads that are not on that path. This imposes a large penalty on kernels with heavy control-flow. The branch divergence metric measures the ratio of divergent branches to the total number of branches present in the kernel for all SMs in the GPU, characterizing kernels with control-flow.

We have implemented the branch divergence metric, using the Lynx instrumentation framework, and present our findings in Figure 7.

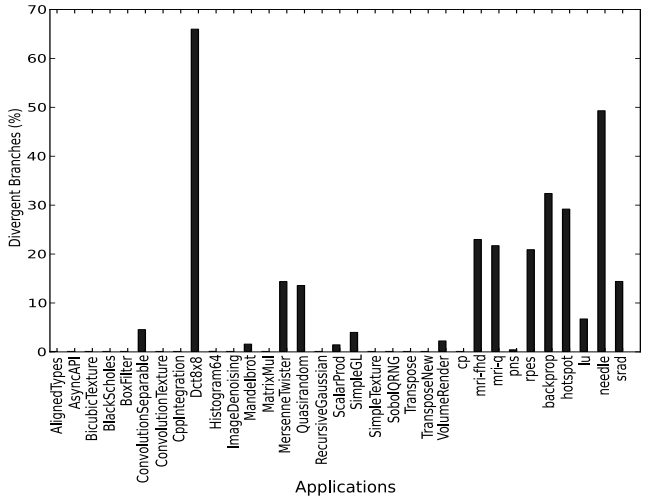

Figure 7: Branch Divergence

Although one might expect to observe an inverse relationship between branch divergence and memory efficiency, less correlation is apparent in practice. Regions with frequent branches and short conditional blocks may have a high incidence of branch divergence but quick re-convergence. Warps executing in these regions consequently exhibit high SIMD utilization, as is the case with *Dct8x8*. Memory instructions located in blocks of the program in which all threads have re-converged therefore have high memory efficiency. Thus, SIMD utilization is more strongly correlated with memory efficiency than the fraction of branches which diverge.

3.4.2.3 Kernel Runtime

The kernel runtime metric is useful in precisely measuring the time it takes for a kernel to execute on a GPU. Although there are other methods for measuring kernel runtimes, such as via source-level assertions, these require participation from the host-side, such as synchronization after the kernel launch, for the measurements to be meaningful [94]. The use of instrumentation enables polling hardware counters on the device to obtain kernel runtimes, which capture precise measurements of multiple events within the execution of a single kernel without including latencies of PCI

bus, driver stack, and system memory.

Our methodology for implementing kernel runtime involves capturing the runtime, in clock cycles, for each thread block and its corresponding SM executing the kernel. This enables us to determine whether the number of thread-blocks and corresponding workloads result in SM load imbalance due to unequal distribution of work among all the thread-blocks for a given kernel. Such fine-grained instrumentation provides useful insights into the GPU thread scheduler's performance via the degree of load balancing it is able to achieve. These insights in turn can provide useful feedback for performance tuning by re-structuring applications, such as the number and size of thread-blocks.

3.4.2.4 Dynamic Instruction Count

The dynamic instruction count metric captures the total number of instructions executed on a GPU. We provide two distinct implementations for this metric. Our first implementation, *counter-per-thread instruction count*, maintains a matrix of counters in global memory with one row per basic block in the executed kernel and one column per dynamic PTX thread. As each thread reaches the end of a particular basic block, it increments its counter index by the number of executed instructions in that basic block. Counters of the same basic-block for consecutive threads are arranged in sequential order to ensure that global memory accesses are coalesced.

Our second implementation is a warp-level instrumentation for the instruction count metric. Since the counter-per-thread instruction count maintains a counter in global memory for *every* dynamic PTX thread, it contributes a significant overhead in terms of memory bandwidth. To reduce this overhead, we implement a *counter-per-warp instruction count*, where a counter in global memory is maintained for every warp instead of every thread. As each warp reaches the end of a basic block, the least active thread in each warp multiplies the active thread count for that warp with the number of executed instructions in that basic block. This product is added to the specific warp's counter.

We use our kernel runtime metric to measure runtimes for the ”most representative”, or the longest-running, kernel with and without our two instruction count instrumentations for selected applications in the CUDA SDK. Normalized runtimes for these kernels are presented in Figure 8.

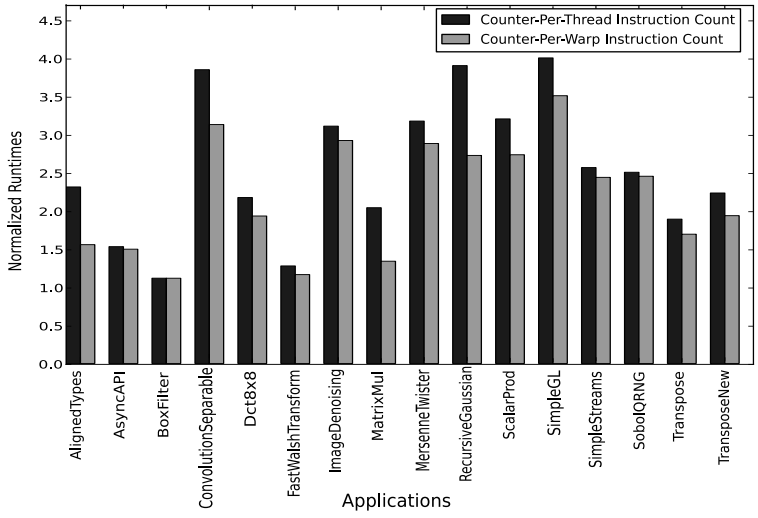

Figure 8: Normalized runtimes of selected applications due to dynamic instruction count instrumentation.

The counter-per-warp implementation consistently outperforms the counter-per-thread instruction count. We attribute this to the lower memory bandwidth overhead resulting from a warp-level instrumentation. However, in some cases, the difference between the two implementations is negligible, such as with *BoxFilter* and *AsyncAPI*, whereas in other cases, the difference is significant, such as with *ConvolutionSeparable* and *RecursiveGaussian*.

We used NVIDIA's Compute Profiler [75] to gain more insight into the L1 cache behavior of these applications. Our findings indicated that the number of global load/store misses in L1 cache, when compared to the number of cache misses with no instrumentation, was between 1.6-1.8$\times$ greater for the counter-per-warp instruction count, and 3-4.5$\times$ greater for the counter-per-thread instruction count, for the *AsyncAPI* and *BoxFilter* kernels. However, in the case of *ConvolutionSeparable* and *RecursiveGaussian*, the number of L1 cache misses for global operations was around

Table 3: Comparison of Lynx with existing GPU profiling tools

FEATURES	*Compute Profiler /CUPTI*	*GPU Ocelot Emulator*	*Lynx*
Transparency (No Source Code Modifications)	YES	YES	YES
Support for Selective Online Profiling	NO	YES	YES
Customization (User-Defined Profiling)	NO	YES	YES
Ability to Attach/Detach Profiling at Run-Time	NO	YES	YES
Support for Comprehensive Online Profiling	NO	YES	YES
Support for Simultaneous Profiling of Multiple Metrics	NO	YES	YES
Native Device Execution	YES	NO	YES

3× greater for the counter-per-warp instruction count and almost 9× greater for the counter-per-thread instruction count. This lends us to believe that certain kernels are more bandwidth-sensitive than others, and for such kernels, the performance gain from the counter-per-warp instrumentation versus the counter-per-thread approach is likely to be more significant.

3.5 Evaluation

We first provide a comparative analysis of Lynx with existing GPU profiling tools, such as NVIDIA's Compute Profiler/CUPTI [75, 76], and GPU Ocelot's emulator. We then look at both the impact of dynamic compilation and the increases in execution times for instrumented kernels, to evaluate the performance of Lynx.

3.5.1 Comparison of Lynx with Existing GPU Profiling Tools

As noted earlier, the core highlights of Lynx's design include transparency, selectivity, and customization. We look at these as well as other features that distinguish Lynx from NVIDIA's profiling tools and GPU Ocelot's emulator. A comparative summary is presented in Table 3.

Lynx provides online instrumentation of applications transparently, i.e. without source code

modifications. Although NVIDIA's Compute Profiler also provides transparency, the performance counters of interest need to be configured prior to the application run and cannot be modified during the execution of the application. Lynx, however, does not require pre-configuration of metrics. As a result, integrating Lynx with online optimizers, such as kernel schedulers or resource managers, is transparently feasible. Additionally, Lynx can attach/detach instrumentation at run-time, selectively incurring overheads only when and where instrumentation is required. This capability is not supported by NVIDIA tools.

Lynx also provides the complementary capabilities of selective and comprehensive online profiling. Users can profile applications at different granularities, such as on a per-thread, per-warp, per-thread-block, or per-SM basis, or can profile all threads for all SMs to capture the complete kernel execution state. NVIDIA tools, however, only provide data for a restrictive set of SMs [75,76]. This limitation results in possible inaccuracy and inconsistency of extrapolated counts, since the data depends on the run-time mapping of thread blocks to SMs, which can differ across multiple runs [75]. It also restricts what one can do with these tools. For example, the thread-block runtimes to SM mappings, implemented via Lynx, informs programmers of potential SM load imbalance. Such fine-grained analysis of workload characterization cannot be obtained by existing NVIDIA tools.

Another distinguishing feature of Lynx is the support for customized, user-defined instrumentation routines, using a C-based instrumentation language. NVIDIA's Compute Profiler/CUPTI provide a specific pre-defined set of metrics to choose from for specific generations of GPUs. User defined instrumentation is not supported. As a result, the memory efficiency metric cannot be obtained by existing NVIDIA tools for NVIDIA GPUs with compute capability 2.0 or higher [75,76]. NVIDIA tools also do not allow certain counter values to be obtained simultaneously in a single run, such as the `divergent_branch` and `branch` counters. This can skew results as multiple runs with likely different counter values may be required to obtain a single metric, such as branch divergence. Lynx, on the other hand, supports simultaneous profiling of multiple metrics in a single

run.

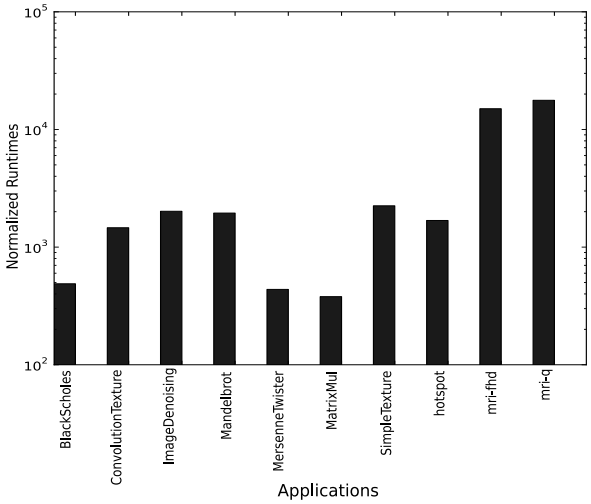

Figure 9: Slowdowns of selected applications' execution on Intel Xeon X5660 CPU via Ocelot's emulator vs execution on NVIDIA Tesla C2050 GPU via Lynx for the memory efficiency metric

Although the Ocelot emulator has most of the capabilities Lynx provides, a core limitation of the emulator is native device execution. Since applications in emulation mode do not run on the GPU, the emulator is unable to capture hardware-specific behaviors and results in being orders of magnitude slower. Figure 9 presents slowdowns of a subset of applications' execution on an Intel Xeon X5660 CPU with hyper-threading enabled (via Ocelot's emulator) versus execution on NVIDIA's Tesla C2050 GPU (via Lynx) for the memory efficiency metric. Applications with short-running kernels were purposely chosen for this experiment since longer-running kernels were prohibitively slow on the emulator.

3.5.2 Performance Analysis of Lynx

3.5.2.1 JIT Compilation Overheads

When using a JIT compilation framework, the compilation time is directly reflected in the application's runtime. To characterize overheads in each step of the compilation pipeline, we divide the total runtime into the following categories: *parse*, *instrument*, *emit*, *moduleLoad*, and *execute*.

Parse is the time taken to parse the PTX module. *Instrument* measures all the various tasks necessary to insert the instrumentation into the application, namely, parsing the C specification, lowering it to the COD IR, translating the IR to PTX, and invoking the pass manager to apply the instrumentation pass over the original kernels. *Emit* is the time spent in GPU Ocelot's PTX emitter, while *moduleLoad* measures the time spent loading the PTX modules into the CUDA driver. Finally, *execute* refers to the execution time of the kernels on the GPU.

Figure 10 shows the compilation overheads for several applications instrumented with the **counter-per-thread instruction count** metric (see Section 3.4.2.4). This metric was chosen because the extent of static code expansion is proportional to the kernel size. The selected benchmark applications show a considerable range of static kernel sizes, ranging from applications with a single, small kernel, such as *MatrixMul*, to those with many, small and medium-sized kernels, such as *TransposeNew* and *pns*, to a single, large kernel, such as *tpacf*.

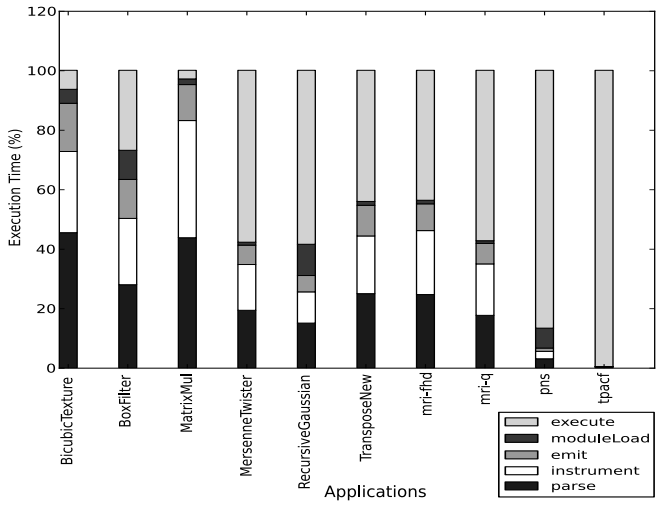

Figure 10: Compilation overheads for selected applications from the CUDA SDK and Parboil benchmark suites, instrumented with dynamic instruction count

The results indicate that the *instrument* overhead is consistently less than the overhead associated with parsing the module but is generally more than the overheads associated with emitting and

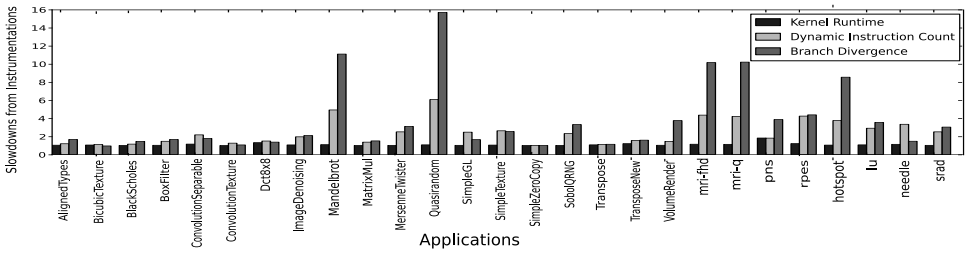

Figure 11: Slowdowns of selected applications due to kernel runtime, dynamic instruction count, and branch divergence instrumentations.

loading the module. Additionally, for applications that either have many small or medium-sized kernels, such as *RecursiveGaussian* and *pns*, or have a single, large kernel, such as *tpacf*, most of the time is spent in executing the kernels on the GPU. This indicates that for longer-running applications, Lynx's JIT compilation overheads are either amortized or hidden entirely.

3.5.2.2 Instrumentation Dynamic Overhead

As shown in the previous evaluation, most of the overhead of instrumentation for longer-running applications is due to executing the instrumented kernels. In this section, we evaluate the overhead of our instrumentation routines on kernel execution times.

Figure 11 presents slowdowns from three instrumentations: kernel runtime, dynamic instruction count, and branch divergence. The kernel runtime metric presents an average slowdown of $1.1\times$, indicating a minimal impact on performance. This is expected since only a few instructions are added to the beginning and end of kernels to perform barrier synchronization and obtain the clock cycle count for all threads in a thread block.

The dynamic instruction count instrumentation incurs both memory bandwidth and computation overheads for every basic block. Therefore, applications with few compute-intensive but potentially large basic blocks, such as *BicubicTexture*, *BlackScholes*, and *Transpose*, experience the least slowdown since memory access costs are amortized. Applications with several short basic blocks, such as *mri-fhd* and *mri-q*, exhibit a much larger slowdown (over $4\times$). Although

Mandelbrot and *QuasirandomGenerator* have a mixture of large and small basic blocks, these two applications exhibit the largest slowdown due to the large number of predicated instructions in their kernels. The dynamic instruction count only takes into account instructions that are actually *executed*. As a result, for every predicated instruction in the kernel, this instrumentation checks whether the predicate is set for each thread to determine if the instruction count needs to be incremented. Consequently, applications that have a large overhead due to branch divergence instrumentation also exhibit a significant slowdown from the dynamic instruction count instrumentation since both of these instrumentations are impacted by the existence of predicated instructions.

Slowdown due to branch divergence varies from as low as 1.05× (*ConvolutionTexture*) for some applications to as high as 15× (*QuasirandomGenerator*) for other applications. This instrumentation depends on the number of branch instructions present in the kernel. Hence, applications that exhibit significant control-flow, such as *Mandelbrot* and *QuasirandomGenerator*, incur the largest performance penalty from this instrumentation while those with little to no control-flow, such as *BicubicTexture* and *ConvolutionTexture*, incur the least penalty.

Figure 12 shows slowdowns of various applications due to the memory efficiency instrumentation.

Once again, we see that certain applications achieve minimal slowdown from this instrumentation while others exhibit slowdowns as high as 30-48×. Since this instrumentation only checks for global memory operations, applications with few global memory operations, such as *BicubicTexture* and *ConvolutionTexture*, exhibit the least slowdown. However, applications with many global memory operations, such as *AlignedTypes*, *QuasirandomGenerator* and *pns*, pay a large performance penalty. Memory efficiency is the most compute-intensive instrumentation we have implemented, with a notable memory bandwidth demand as well. These factors contribute to its high overhead. However, this overhead is to be weighed against the value of information provided by metrics enabled by this type of instrumentation, which is not available via vendor tools. Further, as shown in Figure 9, evaluating such metrics can take several orders of magnitude more time

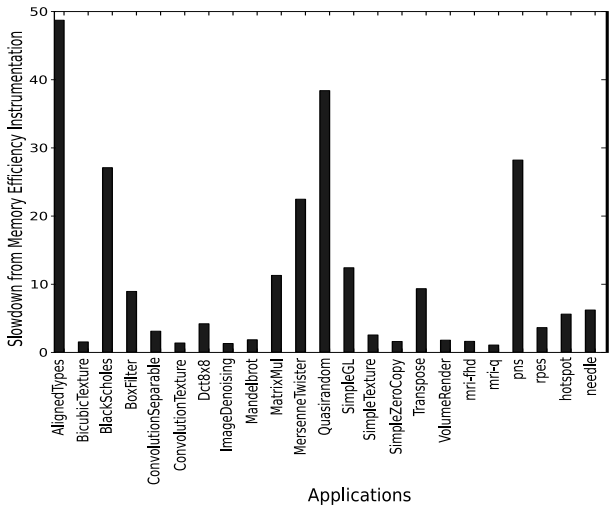

Figure 12: Slowdowns of selected applications due to memory efficiency instrumentation.

using an instruction set emulator.

3.5.3 Optimizing Instrumentation Performance

Certain optimizations can lower the overhead costs of instrumentation. This section describes the information flow analysis used in our framework [38] to reduce the runtime overheads associated with instrumentation, specifically with warp or wavefront-level instrumentation. The primary insight used is that with warp-level metrics, it is critical to identify the program points where thread divergence occurs. This is because basic blocks that exhibit uniform control flow can be easily and more efficiently analyzed via symbolic execution, thus obviating the need to instrument them. By identifying divergent basic blocks, therefore, one can apply instrumentation selectively, thus incurring overheads only where needed. The outcome is an overall reduction in the runtime overheads associated with instrumentation.

The Lynx framework uses a form of information flow analysis, known as taint analysis [70], to track causal dependencies between control-flow instructions and the special registers that keep information about the current thread index. In this analysis, all basic blocks that are branch targets

to conditions dependent on the current thread index are marked as tainted. These basic blocks are identified as possibly exhibiting thread divergence, and thus require instrumentation to ensure precise results. All other basic blocks may be evaluated symbolically.

Our process of performing taint analysis is as follows. We first construct a computation tree. In a computation tree, each node represents the execution of a control-flow statement, such as an *if* statement or the condition of a loop construct, and each edge represents the execution of a sequence of non-conditional statements. As such, a computation tree's edges are representative of its basic blocks. Once a computation tree has been constructed, it is then possible to inspect each node to determine whether it has a reference to the current thread index. If this is the case, the basic blocks represented by the outgoing edges of that node are marked as tainted, as they may exhibit divergent control flow.

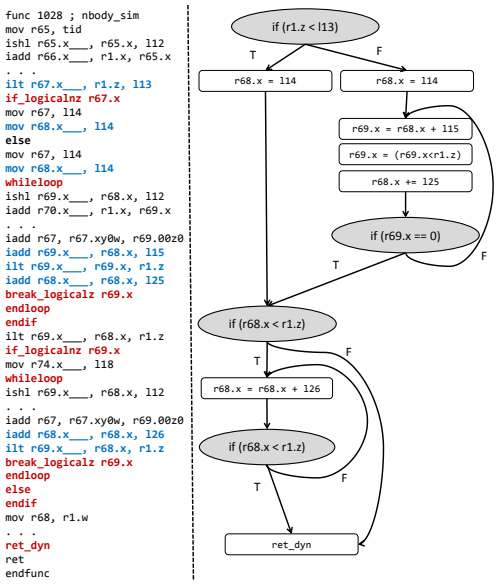

Figure 13: Computation Tree for the NBody Kernel

To illustrate the process, Figure 13 depicts the AMD IL code snippet and the corresponding computation tree for the *NBody* kernel. For purposes of clarity, we have replaced the identifiers of the special registers holding the current thread index with a `tid` variable.

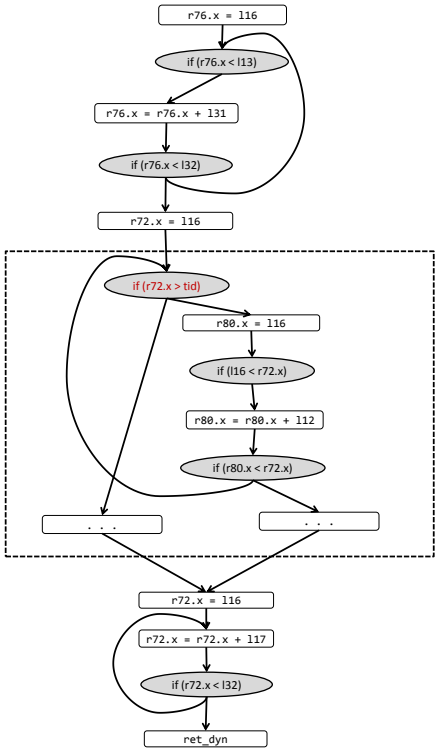

Figure 14: Computation Tree for the LUD Diagonal Kernel

As can be seen from *NBody*'s computation tree, none of the nodes have references to the `tid` variable, thus indicating that this kernel has uniform control flow and can be analyzed entirely via symbolic execution. In Figure 14, we show the computation tree of another kernel, the *LUD diagonal* kernel from Rodinia's LUD implementation [28]. In the case of this kernel, certain parts of the computation tree are dependent on the `tid` variable, while others are not. The basic blocks dependent on `tid` (outlined in the figure) are analyzed via instrumentation, while the remaining ones are analyzed symbolically. We show in our evaluation that our selective instrumentation approach gives us significant reductions in kernel runtime overheads.

3.5.3.1 Symbolic Execution

Symbolic execution [91] is a technique for evaluating a program path-by-path, given its inputs. If all inputs to a program are known prior to execution, it is possible to evaluate each path in the program using concrete input values and determine the precise number of times each basic block is executed. This technique is called concolic (concrete and symbolic) execution [93]. Since the GPU programming model requires kernel arguments and data to be explicitly set on the GPU prior to kernel execution, one can use concolic execution to analyze GPU kernels. However, in the presence of hundreds of threads and possible thread divergence, concolic execution becomes infeasible. Therefore, by identifying sections of the kernel that exhibit uniform control-flow, one can concolically execute such sections under the assumption that a single thread is executing the code, and simply multiply the total number of executed instructions by the number of threads in the grid. Since concolic execution requires only the evaluation of control-flow instructions and instructions that the control-flow operations depend on, it is more efficient than dynamically instrumenting such code, which incurs per-wavefront overhead in terms of both memory bandwidth and computation. Further, instrumentation imposes added runtime overheads due to the dynamic generation of an updated binary comprised of the instrumented kernels. Note, however, that a possible contributor to high overhead with concolic execution is loop unrolling. With concolic

execution, loop conditionals are continually evaluated until the loops are completely unrolled. However, in all the kernels we evaluated for this study, the overhead from loop unrolling was insignificant.

3.5.3.2 Evaluation of Instrumentation Overheads

We demonstrate the benefits of combining symbolic execution with dynamic instrumentation in reducing kernel runtime overheads, and also show that symbolic execution for non-divergent code is much more efficient than runtime instrumentation.

All experiments for this section are performed on a system with an Intel Core i7 running Ubuntu 12.04 LTS x86-64, equipped with an AMD Radeon HD 7770 GPU. Benchmark applications for experiments are chosen from the AMD OpenCL SDK [6] and the Rodinia Benchmark Suite [28]. Seven applications are used in this study, described in Table 4. They are selected to ensure good coverage in terms of control-flow irregularity, since the approach using symbolic execution is dependent on the presence of control-flow regularity. Consequently, the selected applications (1) have no control-flow statements (*MatrixTranspose*, *FastWalshTransform*), (2) have uniform control-flow (*NBody*), (3) have mostly divergent code (*Reduction*, *BitonicSort*), and (4) have both divergent and uniform code sections (*LUD*).

Table 4: Benchmark applications evaluated for instrumentation overheads

Benchmark	**Domain**	**Source**
Bitonic Sort	Sorting	AMD SDK
FastWalsh Transform	Signal Processing	AMD SDK
LUD (diagonal kernel)	Linear Algebra	Rodinia
Matrix Transpose	Linear Algebra	AMD SDK
Nearest Neighbor	Data Mining	Rodinia
NBody Simulation	Physics	AMD SDK
Reduction	Sorting	AMD SDK

Figure 15 presents the kernel runtime overheads that result from applying dynamic instrumentation to all kernels, vs. selectively instrumenting kernel code based on information flow analysis

and symbolic execution. We took an average of ten runs for each kernel, with and without each instrumentation, and the final overhead results are averaged for both of the two GPU metrics we implemented, activity factor and memory intensity. Given that both activity factor and memory intensity are warp-level and counting-based instrumentations, the overheads for each of them followed the exact same trend for all of our application kernels.

Note that the kernel runtime slowdown for three of the seven kernels, namely *FastWalshTransform*, *LUD*, and *NBody*, are quite high, ranging from 8-24×. This high overhead is attributed to the presence of many, small dynamic basic blocks in these kernels. Since both activity factor and memory intensity metrics contribute per-basic block overhead in terms of memory bandwidth and computation, kernels with a large number of dynamic basic blocks exhibit a larger slowdown. In addition, we believe that our implementation of these metrics also contributes to the high slowdown associated with dynamic instrumentation of all kernels in general. Due to the lack of special instructions that can provide us with wavefront-level information in the AMD IL, we chose to use atomics and memory barriers on local memory to obtain this data. We went with a non-optimal implementation choice because the HSA IL, which will replace the AMD IL for future AMD systems, has support for such intrinsics. Therefore, with HSA IL, we no longer will need to rely on atomic and barrier operations to implement these metrics. Note that irrespective of the implementation details of the instrumentation, the program analysis techniques we present in this work to perform more fine-grained instrumentation of GPU kernels are orthogonal and complementary in reducing the overheads. The take-away is that dynamic instrumentation always has some runtime overhead associated with it, so any technique that can help reduce or eliminate the need for instrumentation altogether, while precisely characterizing the desired runtime behavior of the application, is beneficial.

As expected, kernels that have either uniform or no control-flow (i.e., *BitonicSort, MatrixTranspose, NBody,* and *FastWalshTransform*), exhibit almost no kernel runtime slowdown. This

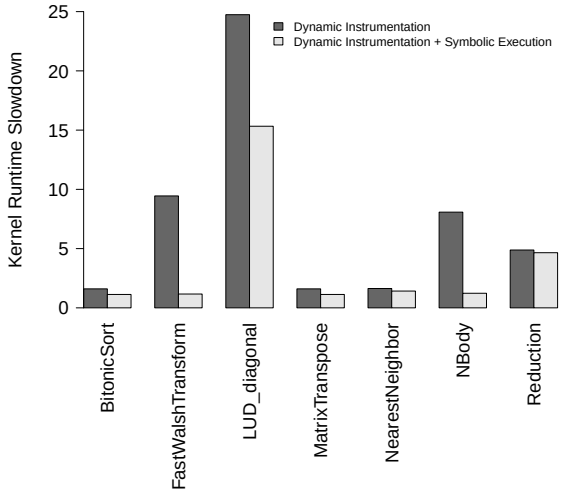

Figure 15: Kernel Runtime Slowdown due to Instrumentation

is because these kernels are evaluated entirely via symbolic execution. *NBody* and *FastWalsh-Transform* benefit most from symbolic execution, because not only do these kernels have uniform control-flow, but they also have a large number of dynamic basic blocks. Since the activity factor instrumentation operates at basic-block level, kernels with many short basic blocks pay a higher runtime price than those with a few large basic blocks. *MatrixTranspose, BitonicSort,* and *NearestNeighbor* fall into the latter category.

Both *Reduction* and *LUD* have a combination of divergent and uniform code segments. However, the benefit of symbolic execution is much greater for the *LUD* kernel than the *Reduction* kernel. This correlates directly with the extent of divergent code present in these kernels. *LUD* has a good mix of divergent and uniform code segments while *Reduction* has a much larger divergent code section, with only a few instructions that fall in the non-divergent code section. As a result, we are able to symbolically execute more of the *LUD* kernel versus the *Reduction* kernel. Note that despite the *LUD* kernel having a more substantial uniform code segment than the *Reduction* kernel, it is still smaller than its divergent code section. Nevertheless, we are still able to get almost a 38% improvement by using symbolic execution in combination with dynamic instrumentation for this

kernel.

3.6 Related Work

Pin [62] is a dynamic instrumentation system for CPU application binaries that supports multiple CPU architectures, such as x86, Itanium, and ARM. Just like Pin, Lynx also supports the creation of portable, customized program analysis tools. However, unlike Pin, Lynx targets data-parallel applications and enables the execution of such programs on heterogeneous back-end targets, such as GPUs and multi-core CPUs.

NVIDIA's Compute Visual Profiler [75], *CUDA Profiling Tools Interface*, CUPTI [76], and AMD's CodeXL [7] were released to address the profiling needs of developers of GPU compute applications. Each provides a selection of metrics to choose from by reading performance counters after applications have run. CUPTI enables the creation of profiling tools for CUDA applications via APIs that perform source code interjection. Although in some cases these utilities provide similar information that can be obtained via GPU Lynx, none of them support the ability to insert customized, user-defined instrumentation procedures, nor support selective online profiling. For example, with our framework, users can profile applications at different granularities, such as on per-thread, per-wavefront, per-thread-block, or per-core basis. Further, as discussed in this work, our framework allows for fine-grained instrumentation of selective code sections. Finally, since our framework supports the ability to toggle profiling on and off while the application is running, we can support online, profile-driven optimizations, code transformations, and resource management policies.

The TAU Performance System [94] provides profiling and trace analysis for high-performance parallel applications. TAU, however, relies on source instrumentation and library wrapping of the CUDA Runtime/Driver APIs to obtain measurement before and after CUDA API calls via call-backs or events. TAU also does not offer the flexibility to end-users to define their own customized instrumentation routines and relies heavily on the device manufacturer to provide support for events

and callbacks. Furthermore, unlike TAU, by obtaining kernel execution measurements via Lynx's dynamic instrumentation system, there is no participation needed from the host to perform the GPU-related measurement.

GPU simulators and emulators [14, 30, 51], visualization tools built on top of such simulators [10], and performance modeling tools [13, 105] are generally intended for offline program analyses to identify bottlenecks and predict performance of GPU applications. Many metrics of interest typically obtained through simulation may also be obtained through instrumentation, particularly those metrics reflecting application behavior. Thus, Lynx is one viable tool to drive research efforts such as the aforementioned. Hardware interactions may also be observed through instrumentation, provided care is taken to avoid perturbing the very effect being measured. With Lynx's selective and efficient profiling tools, the same levels of detail can be captured while natively running applications on the GPU, achieving notable speedups over CPU execution.

Sophisticated program analysis techniques for GPU computing have been studied extensively in dynamic compilation frameworks, such as GPU Ocelot [33], and code generation frameworks, such as Caracal [34] and [40]. GPU Ocelot [33] is a dynamic compilation framework designed to map the explicit data-parallel, GPU execution model onto multi-threaded, many-core x86 processors, leveraging the Low Level Virtual Machine (LLVM) [55] code generator. Grewe et al. [**?**], on the other hand, designed a code generation framework to automatically generate OpenCL code from data-parallel OpenMP GPU programs. These works make use of intricate program analysis techniques. To the best of our knowledge, our work is the first to apply information flow analysis and symbolic execution to develop a more efficient instrumentation framework.

3.7 Chapter Summary

This chapter presents the design and implementation of Lynx, a dynamic instrumentation toolchain that provides *selective*, *transparent*, *customizable* and *efficient* instrumentation for GPU computing platforms. Lynx's goal is to facilitate the creation of binary instrumentation tools for data-parallel execution environments, a novel capability that no existing tool available to the research community provides. Auto-tuning and optimization tools can be built on top of Lynx's dynamic instrumentation engine to improve program performance at runtime. The online feedback capability provided by Lynx also makes it a viable candidate to drive run-time resource management schemes for high-performance GPGPU-based clusters. Monitoring and execution analysis tools can be integrated with Lynx to run natively rather than on a simulator, achieving several orders of magnitude speedup. Furthermore, Lynx enables the creation of correctness checking tools that provide insights into both functional and performance bugs to improve programmer productivity.

CHAPTER IV

LUMINAR: INSTRUMENTATION-DRIVEN RESOURCE MANAGEMENT ON INTEGRATED CPU/GPU PLATFORMS

4.1 Introduction

Rapid, predictive data analytics, including graph analysis, are important techniques in social, scientific, and retail industries. This has given rise to extensive work on efficiently running such applications on today's server systems– new techniques for algorithm parallelization and graph partitioning [58, 61], specialized machine learning and graph frameworks [71], and runtime support to accelerate analysis using GPU platforms [16, 88, 102].

To this body of work, we contribute new methods for accelerating graph applications on next generation servers with integrated GPUs [4]. Integrated GPUs are power-efficient and can directly access the same large volumes of main memory as those used by CPUs. This direct access increases the memory available for storing data, avoids costly data movement between host and GPU memories, and enables fine-grained resource sharing between CPU and GPU devices. The importance of integrated CPU-GPU processors is evident from the recent rise in micro-servers, a market favoring energy efficiency, where e.g., HP's Moonshot servers accelerate hosted desktops using low power AMD Opteron processors (11W TDP) and integrated Radeon graphics [3].

There are substantial technical challenges that need to be addressed before applications can benefit from integrated GPU platforms:

1. Device-application match. GPUs benefit compute-intensive applications that feature minimal synchronization, uniform control flow, and regular memory access patterns. Applications exhibiting large regions of serialized code or irregular control-flow are better suited to the CPU's execution

model. As shown in Figure 16, regular, compute-intensive applications like Black-Scholes (BS) and K-Means (KM) exhibit 5-23× speedup on the GPU over the CPU, while irregular applications like Page-Rank (PR) can be up to 3× faster on CPUs, compared to GPUs, for certain inputs. Therefore, the benefit an application derives from a device depends upon its control-flow irregularity and memory bandwidth requirements.

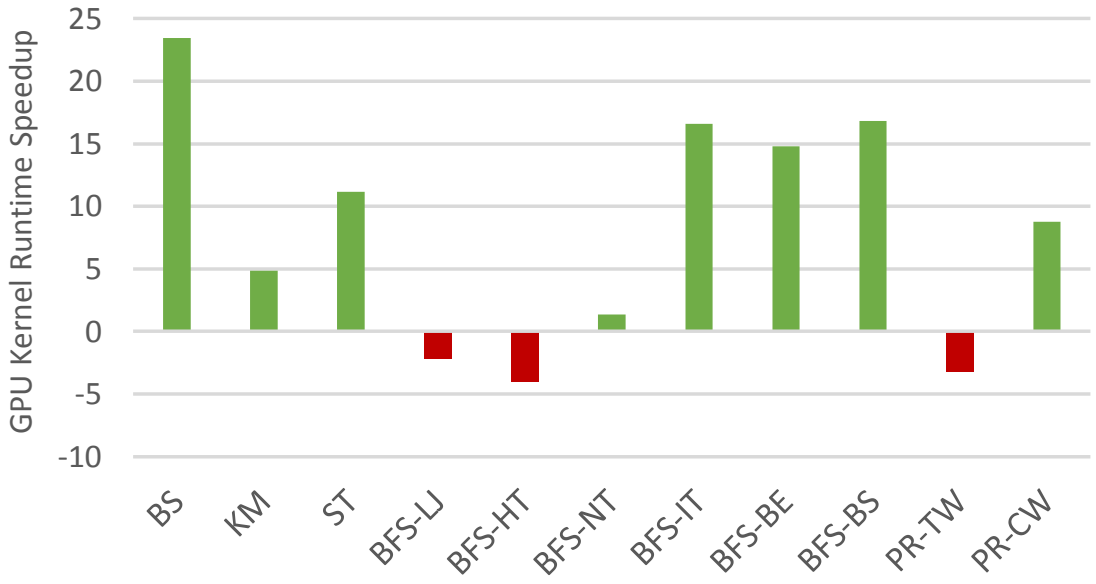

Figure 16: Kernel runtime speedup on GPU versus multi-core CPU execution for selected workloads.

2. Input-dependent, irregular workloads. The performance of applications with input-dependent and irregular runtime behavior is difficult to predict statically, with graph algorithms, such as Breadth-First Search (BFS) and Page-Rank (PR), being classic examples. This behavior is highlighted in Figure 17, which depicts GPU-level thread *activity factor* across BFS iterations for *two distinct input graphs*, `LiveJournal` social network and `Italy` street network. Activity factor [51] is a runtime metric measured as a percentage of threads actively performing computation over the total number of available threads. Activity factor on `LiveJournal` exhibits significant variance across runs, while it is relatively constant for the `Italy` road network. Interestingly,

BFS on `LiveJournal` has poor performance on GPUs (Figure 16), compared to the `Italy` graph. Unfortunately, achieving high performance on heterogeneous architectures today involves effort-intensive tuning for each input and manual evaluation of alternative code versions for the best performance. Luminar, on the other hand, uses fine-grained runtime metrics, such as activity factor, to choose an alternative code version, *seamlessly and transparently at runtime*, to improve performance of irregular graphs, such as `LiveJournal`.

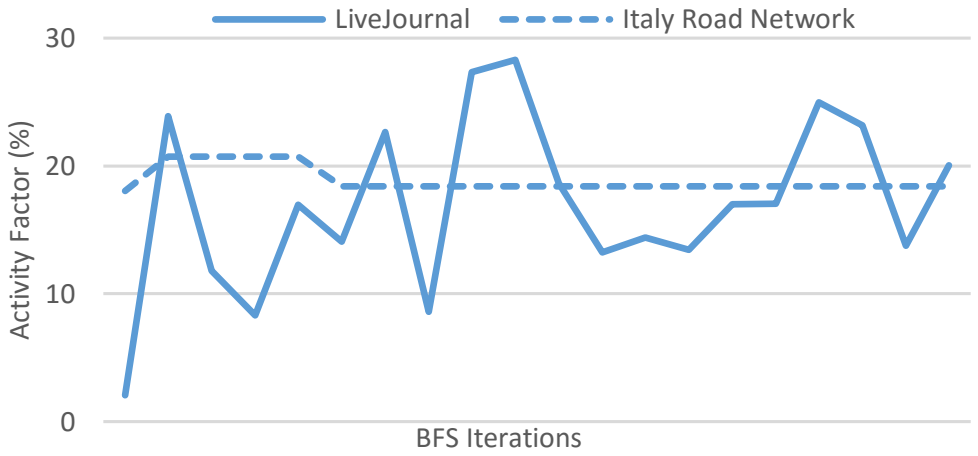

Figure 17: Thread activity varies widely across BFS iterations on two distinct graphs: LiveJournal social graph, and Italy street network.

3. Memory contention. Integrated GPUs share the same physical memory with the CPU. While shared memory reduces data transfer overheads and offers fine-grained opportunities for resource sharing, it also exposes applications to potential contention issues. In multi-user settings, memory contention can degrade performance of co-scheduled memory-bound applications. For example, when co-executing memory-bound kernels on the CPU and GPU devices, workloads can exhibit as much as 60% performance degradation (Section 4.3.2).

Prior work has tackled some of these issues in isolation, primarily for discrete GPUs (GPUs attached externally via PCIe). For example, static profiling and empirical auto-tuning have been used for CPU-GPU scheduling [46, 78, 83], and to characterize applications [14, 30, 51]. Online

profiling has been used when running a single application across CPU and GPU cores [47], but these techniques do not address challenges posed by irregular, input-driven applications, and the memory contention [65] in integrated GPUs.

We present *Luminar*, the first system to use runtime measures for improving application performance on integrated GPUs. It offers particular benefits to irregular, input-dependent workloads like graph applications. Luminar makes the following technical contributions:

- *Dynamic instrumentation* inserts code snippets into the application execution path at runtime, to gain insights into runtime application behavior. Fine-grain metrics characterizing applications include (i) *activity factor*, which captures GPU-level thread activity, and (ii) *memory intensity*, which captures memory usage patterns. Low overheads are achieved by the use of *selective* instrumentation, and time-varying runtime behavior is handled by *continuous, repeated* profiling.

- *Runtime resource management*, guided by dynamic application metrics, automatically determines which application codes are run where – on CPUs vs. GPUs – without user intervention, tuning, or similarly cumbersome developer actions.

- *Extensive experimental studies*, driven by real applications and data sets, evaluate both the performance and energy efficiency of integrated GPU platforms, demonstrating the effectiveness of dynamic instrumentation and runtime resource management in improving application performance and system throughput.

Our evaluation shows promising results when *Luminar*'s approach is applied to irregular workloads such as graph applications. Luminar can use instrumentation-driven profiling to select the best GPU code version among multiple code variants for a given input graph, achieving 40% to $2\times$ performance improvement over a static choice of a particular implementation. In addition, Luminar determines device *affinity* (CPU vs. GPU) for distinct input graphs, dynamically assigning

work to the preferred device. The outcome is a *device-affinity, contention-aware* scheduling that improves the performance of applications running concurrently on CPU and GPU cores. With this scheduler, Luminar gains 21-81% in system throughput and 3-60% in energy efficiency compared to the best baseline policy.

4.2 Luminar Overview

Luminar uses dynamic instrumentation to perform profile-guided optimizations and drive scheduling decisions on integrated GPUs. Unlike discrete GPUs, which transfer data between CPU and GPU via an external bus like PCIe, integrated GPUs have the CPU and GPU on a single die and share the same physical memory.

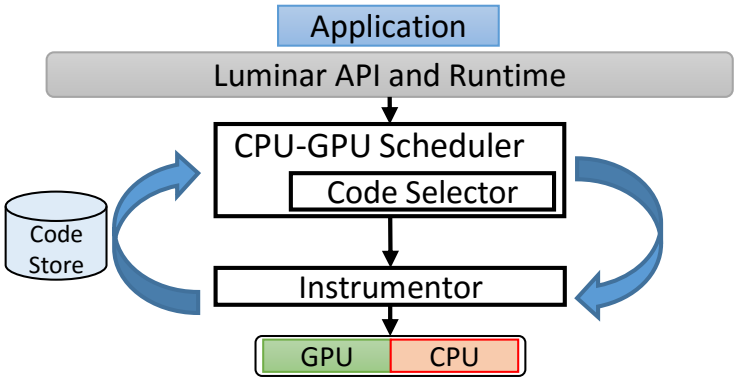

Figure 18: Luminar Architecture

Figure 18 provides an architectural overview of Luminar's four primary components– a *CPU-GPU Scheduler*, an *Instrumentor*, a *Code Store*, and a *Code Selector*. Applications interface with Luminar's runtime using the API shown in Listing 4.1. Luminar's API leverages the OpenCL runtime to provide cross-platform support for executing computations across heterogeneous platforms. Luminar maintains a single OpenCL context for both CPU and GPU devices, and a separate command-queue for each device. The Luminar API allows end-users to register their applications

with the runtime. This gives Luminar's runtime complete control over dispatching work and transferring data between heterogeneous devices, without requiring any programmer intervention or even rebuild of application binaries. Luminar uses the *Code Store* to archive the original and instrumented kernel binaries, in order to avoid compilation overheads when the same application is executed multiple times.

Listing 4.1: Luminar API

```
/* Applications register kernels and I/O buffers. Original and
   instrumented kernel binaries are generated and saved for future use.
    */
void registerKernel(string kernelId)
void registerBuffer(string bufferId, cl_mem_flags flags, cl_int size,
   void *hostPtr);

/* Applications retrieve kernel and data buffer objects using
   registered ids. */
cl_kernel getKernel(string kernelId);
cl_mem getBuffer(string bufferId);

/* Applications specify unique kernel-input ids when acquiring and
   releasing devices. Luminar assigns the appropriate device queue to
   the application, based on its profile. */
void acquireDevice(string kernelId, string inputId, cl_command_queue *
   queue);
void releaseDevice(string kernelId, string inputId, cl_command_queue *
   queue);
```

When a kernel has to be executed, Luminar invokes the *CPU-GPU Scheduler* to map the application on either the CPU or GPU. The scheduler uses the online profile of the kernel to determine suitability of running the application on the CPU versus the GPU. Next, the scheduler uses the runtime state of the device to determine when the application should be scheduled on the chosen device (CPU or GPU). If the particular kernel profile does not exist or is out-dated, the *Instrumentor* selectively instruments the kernel code and measures the application's runtime behavior on the GPU. Kernel profiles are maintained as a JSON dictionary, and are updated continuously, at repeated intervals, to accommodate time-varying runtime behaviors.

The *Code Selector* provides the ability to further perform application-specific acceleration. It can take additional user-specified actions based on the kernel profiles. To demonstrate its viability, we have implemented a *Code Selector* for the Breadth-First Search (BFS) application. It dynamically chooses the best GPU kernel code between two state-of-the-art BFS implementations, based on the application's thread activity on the GPU. In this implementation, the end-user is responsible for registering multiple kernel implementations with Luminar.

4.3 Dynamic Instrumentation

Luminar performs dynamic instrumentation via an existing open-source library – GPU Lynx [35], which embeds into kernels instrumentation code to measure customized, user-defined metrics. Such runtime instrumentations can be used for debugging, correctness checking, and profile-driven optimizations. For this work, we implement a runtime that provides the following features: (i) an OpenCL interposer that integrates with GPU Lynx to make instrumentation *transparent* to the applications, (ii) compiler passes to insert instrumentation for two runtime metrics, *activity factor* and *memory intensity*, and (iii) a modified instrumentation engine to *selectively instrument* only a subset of the wavefronts and specific basic blocks in a kernel. Thread divergence, measured by *activity factor*, and memory bandwidth, measured by *memory intensity*, are known performance limiters on the GPU. Therefore, Luminar uses these two metrics to make intelligent scheduling

decisions at runtime. With these new capabilities, Luminar provides *online*, *transparent*, and *low overhead* mechanisms to profile applications.

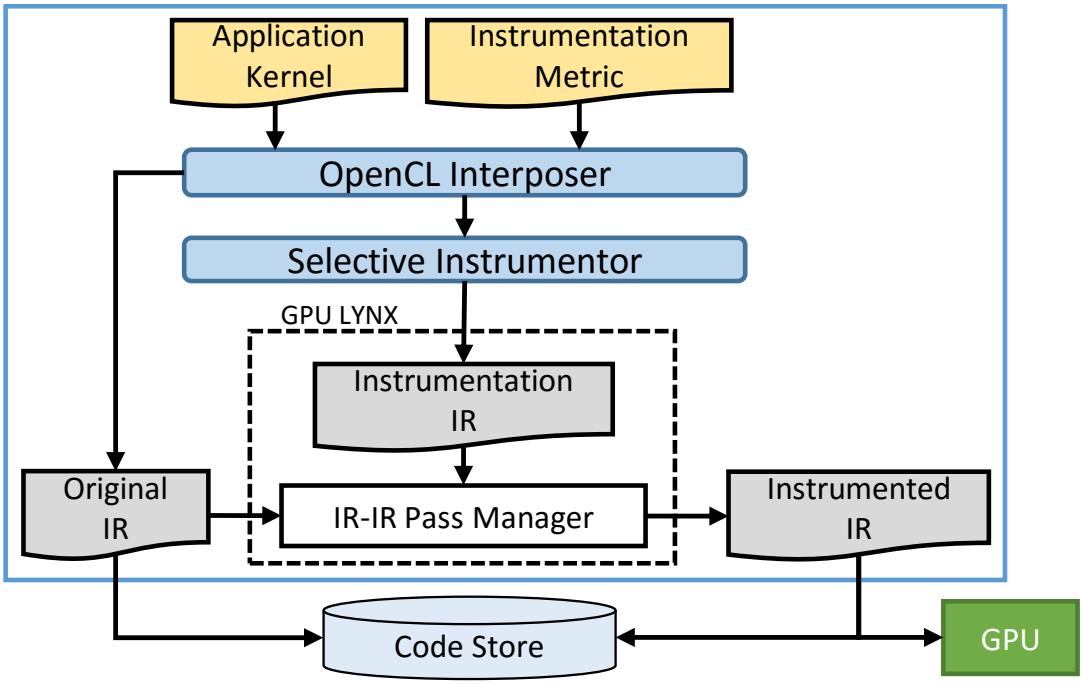

Figure 19: Luminar's *Instrumentor* Component

Figure 19 shows Luminar's instrumentation framework. The instrumentation metric specification is provided to the system along with the original application kernel. The specification defines where and what to instrument. Instrumentation can be defined at the kernel level, basic block level, or the instruction level.

Luminar's OpenCL Interposer orchestrates the generation of the final instrumented kernel by enlisting the *Selective Instrumentor* and GPU Lynx's IR-IR (intermediate representation) transformation pass manager. The *Selective Instrumentor* refines the original instrumentation metric, specifying a subset of wavefronts and basic blocks as candidates for instrumentation.

Two instrumentation metrics are implemented for this work, *activity factor* [51] and *memory*

intensity [51]. These metrics characterize an application's GPU affinity and its memory boundedness, respectively. Luminar generates kernel profiles based on these metrics at runtime. As application characteristics may depend on input data, Luminar generates application profile for each distinct pair of application kernel and input. In other words, if the same kernel is executed with K distinct inputs, there are K distinct profiles for that kernel.

Listing 4.2: Instrumentation Specification

```
ON_BASIC_BLOCK_ENTRY:
  uint wid = get_global_id(0)/WAVEFRONT_SIZE;
  uint tid = get_local_id(0);
  actTid = getActiveThreadInWavefront();
  activeThreadCount = activeThreadsInWavefront();
  if(tid == actTid && wid % N == 0) {
    uint offset = (wid/N)*2;
    /* counters for activity factor metric */
    instBuffer[offset] += activeThreadCount * basicBlockSize();
    instBuffer[offset+1] += WAVEFRONT_SIZE * basicBlockSize();
    /* counters for memory intensity metric */
    instBuffer[offset+2] += activeThreadCount * memOpsInBasicBlock();
    instBuffer[offset+3] += activeThreadCount * basicBlockSize();
  }
```

Listing 4.2 depicts sample instrumentation for activity factor and memory intensity, occurring at the basic block level and focused on the behavior of a *wavefront* rather than of independent threads. A *wavefront*, in OpenCL terminology, is the smallest schedulable unit on a GPU core. Threads within a wavefront execute the same instruction in lockstep fashion.

To calculate activity factor and memory intensity, instrumentation code is inserted at the beginning of each basic block to obtain the *active thread count* in each wavefront. The instrumentation

Table 5: Selected workloads for Luminar

Workload	Description
BS	Black-Scholes on 200 M options
KM	K-Means on 20 M, 10-dim points, 50 centers
ST	Stencil, 100 iterations on 512x512x64 grid
BFS-LJ	BFS on LiveJournal graph (N=5 M, E=69 M)
BFS-HT	BFS on Higgs-Twitter graph (N=0.45 M, E=15 M)
BFS-NT	BFS on NLPKKT160 graph (N=3.5 M, E=110 M)
BFS-IT	BFS on Italy road graph (N=6.7 M, E=7 M)
BFS-BE	BFS on Belgium road graph(N=1.4 M, E=1.5 M)
BFS-BS	BFS on Berk-Stan graph (N=0.69 M, E=7.6 M)
PR-TW	Page-Rank on Twitter graph (N=33 M, E=282 M)
PR-CW	Page-Rank on Clueweb graph (N=100 M, E=2 B)

buffer is allocated prior to kernel execution to store the desired number of counters per wavefront in GPU's global memory. After kernel execution, activity factor and memory intensity is calculated by aggregating the counters for all wavefronts. Luminar provides users with a configurable parameter, *N*, which enables the system to selectively instrument every N^{th} wavefront.

All our experiments were conducted on an AMD APU desktop with four 3.8GHz CPU cores, an 800MHz integrated GPU, and 32GB DRAM. Details about the applications and datasets appear in Table 5. In all speedup plots, we compare GPU performance to applications running on all four CPU cores.

4.3.1 Activity Factor

This metric characterizes how well an application is utilizing a GPU's SIMD parallel execution model. It measures the degree of control-flow irregularity present during application execution. When threads within a wavefront diverge due to a data-dependent control flow statement, the wavefront serially executes each branch path taken, disabling threads that are not on that path. Threads that are not disabled on a given path are considered to be *active*. Mathematically, activity factor is defined as:

$$AF = \frac{\text{instructions executed by all active threads}}{\text{instructions executed by all launched threads}} \quad (1)$$

Since control-flow irregularity is often data-dependent, activity factor for an application may vary with different inputs. A high activity factor indicates uniform or no control-flow, which is ideal for GPU execution, whereas a low activity factor indicates a lot of control-flow irregularity.

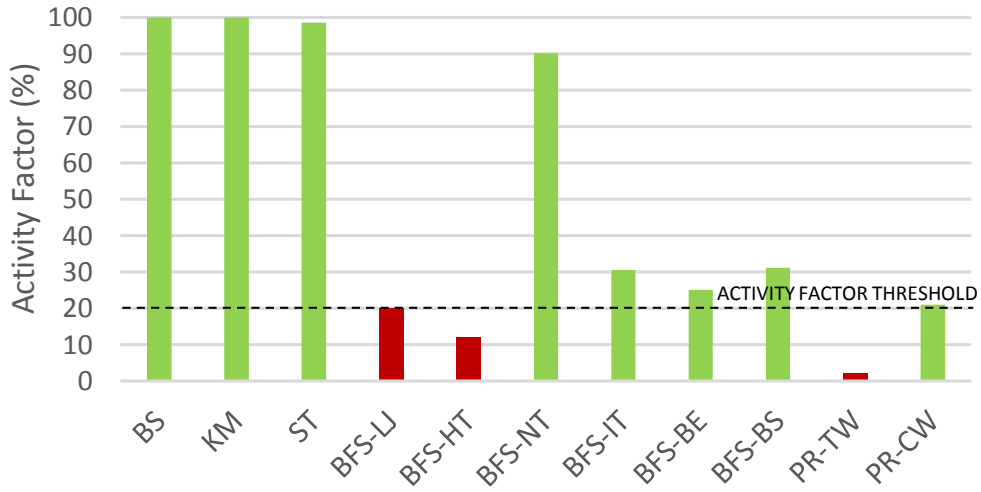

Figure 20: Average activity factor of workloads. Applications with average values below the activity factor threshold do not benefit from GPU execution (as shown in Figure 16).

Figure 20 shows the activity factor for multiple applications measured by Luminar. Note that a high activity factor correlates to good performance on the GPU (see Figure 16). However, the amount of GPU speedup depends on other architectural and application-specific attributes as well. For instance, a GPU with a large number of SIMD cores is likely to provide greater benefit to an application with a high activity factor, over one with fewer cores. Additionally, application-specific characteristics, such as its use of synchronization primitives, also impact how well an application can exploit SIMD parallelism. For example, `BFS-NT` has only 34% GPU speedup, even though the activity factor is high. The heavy use of atomics in this implementation contributes to high overheads for graphs with large and varying frontier sizes, such as *BFS-NT*.

As a result, we use calibration runs to determine the *activity factor threshold*, the minimal value at which applications benefit from GPU execution, for our hardware configuration. Note this profiling step is required only once for a given platform to account for specific architectural attributes of the underlying CPU and GPU devices. We also use a diverse set of micro-benchmarks from the AMD SDK [6], Parboil [44], and Rodinia [28] benchmark suites in our calibration runs, consisting of workloads with varying degrees of control-flow irregularity and synchronization primitives. Based on these calibration runs, Luminar uses an activity factor threshold of 20% for our given configuration to determine whether an application has affinity to the GPU versus the CPU.

4.3.2 Memory Intensity

This metric characterizes the extent to which an application is memory-bound versus compute-bound, and is defined as:

$$MI = \frac{\text{total executed global memory instructions}}{\text{total executed instructions}} \tag{2}$$

Global memory instructions refer to read/write accesses to the GPU's global memory subsystem. For integrated GPUs, this memory resides on the CPU. The memory intensity metric can be used to determine which two applications should or should not be run concurrently to avoid memory contention. Figure 21 depicts the memory intensity metric measured by Luminar for different applications. A high memory intensity implies that an application is memory-bound versus compute-bound. Once again, calibration runs are performed to determine the memory intensity threshold for our given configuration, at 10%.

To understand the correlation between memory intensity and contention, we measure the slowdown introduced by different types of workloads concurrently running on the CPU and GPU. Figure 22 shows the CPU and GPU kernel runtime slowdowns in three settings: concurrently running compute-bound kernels (those with values less than the memory intensity threshold), running a

mix of compute-bound and memory-bound kernels, and concurrently running memory-bound kernels (those with values greater than the memory intensity threshold). As expected, the concurrent execution of compute-bound kernels has the least impact on performance, and the performance degradation is worst when memory-bound kernels are run concurrently on the CPU and GPU, with total slowdown of up to 60%.

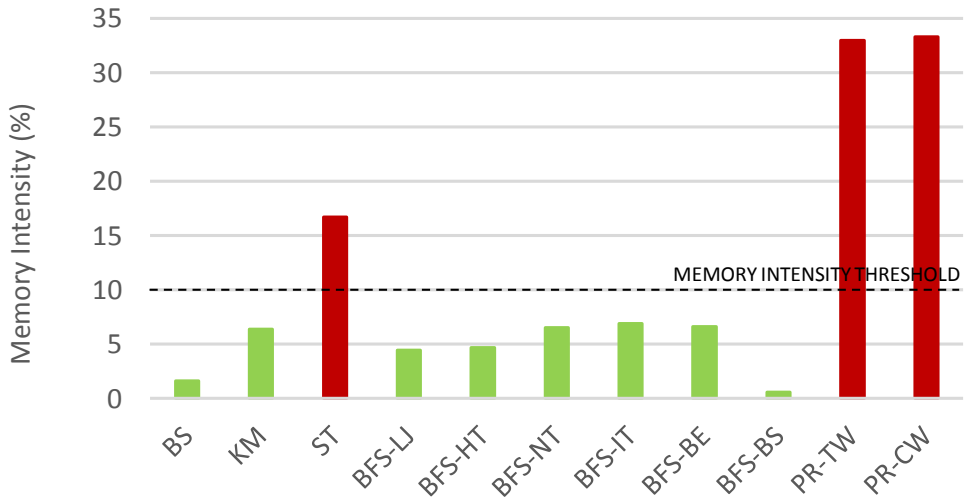

Figure 21: Memory intensity of selected workloads. The memory intensity threshold is 10%.

4.3.3 Performance vs. Accuracy

Dynamic instrumentation can have high overhead (Figure 23), as it incurs both memory bandwidth and computation overheads for every basic block. Therefore, applications with few compute-intensive but large basic blocks, such as Black-Scholes, experience the least slowdown since memory access costs are amortized. Applications with several short basic blocks, such as K-Means, exhibit a much larger slowdown.

Luminar uses *selective* instrumentation to reduce these overheads. In our current implementation, we instrument 5% of all wavefronts (every 20th wavefront), and use static program analysis to only instrument basic blocks in loops. With selective instrumentation, we observe the overheads

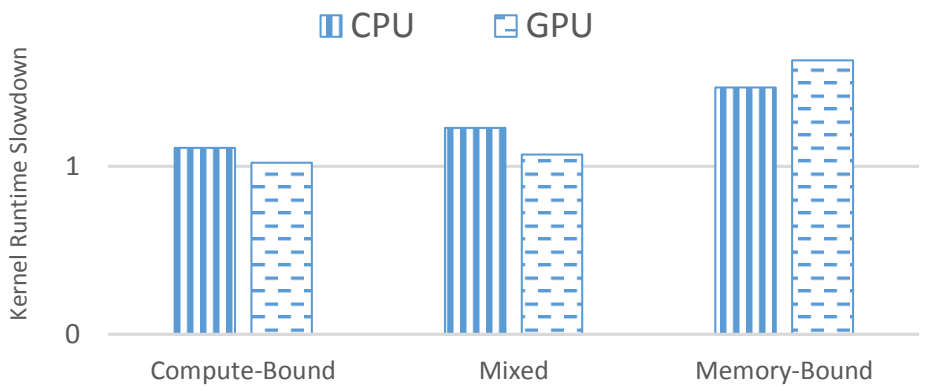

Figure 22: Kernel runtime slowdown due to memory contention in three settings: (a) co-executing compute-bound kernels, (b) co-executing mix of compute-bound and memory-bound kernels, and (c) co-executing memory-bound kernels. Lower is better.

of running Luminar to be less than 2% for most applications.

For *regular* applications, which exhibit mostly uniform control-flow, selective instrumentation is as accurate as full instrumentation because all wavefronts follow a similar execution pattern. For *irregular* applications like BFS and Page-Rank, selective instrumentation improves performance at only a small loss in accuracy. In the case of Page-Rank, we use activity factor to determine its affinity to the GPU versus the CPU (discussed in Section 4.4.1). Our experiments show that the loss in accuracy of activity factor, due to selective instrumentation, is negligible for the Twitter graph (less than 1%), and at most 5% for the Clueweb graph. In the case of BFS, we use the *variance* in activity factor to characterize graph regularity (discussed in detail in Section 4.4.2). When running BFS from 10,000 randomly selected source nodes on each of our input graphs, we compare full instrumentation activity factor variance results with selective instrumentation activity factor variance over the first five iterations. We are able to characterize graphs correctly approximately 99% of the time with our selective instrumentation methodology. We noticed that other strategies, such as instrumenting slightly more iterations or randomly choosing which iterations to instrument, do not provide additional benefits.

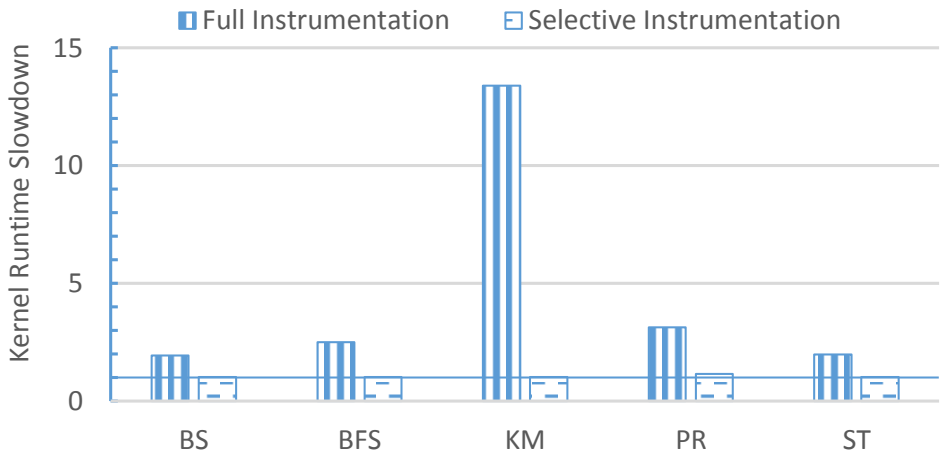

Figure 23: Slowdown due to overheads of full instrumentation, i.e. all wavefronts instrumented, versus selective instrumentation, i.e. 5% of wavefronts instrumented.

4.4 Profile-Guided Optimizations

This section describes how activity factor can guide profile-guided optimizations, such as dynamic mapping and runtime code selection for graph applications. Real world graphs are irregular, i.e., the number of edges per vertex is skewed [43]. This irregularity can lead to control flow divergence and, hence, poor performance of the application on GPUs. An application's performance on the GPU, therefore, depends not only on its implementation but also on its input data. This highlights the need for the dynamic profiling and mapping approach pursued by Luminar.

4.4.1 Dynamic CPU-GPU Mapping

We use Page-Rank to show that Luminar can accurately characterize an application's GPU performance on different inputs and then use the information to decide whether to map the application onto the GPU or the CPU.

Activity factor can characterize a graph application's performance on the GPU. Activity factor captures the amount of work performed by the GPU threads as well as the degree of load-imbalance in the input graph. Figure 20 shows that the average activity factor for the Twitter graph

is 1%, compared to that of 21% for the Clueweb graph. A higher activity factor is indicative of more efficient use of the GPU's SIMD parallelism. The first two bars in Figure 24 shows the time to complete Page-Rank on both the graphs, one after the other, when executed on the CPU and GPU, respectively. As predicted by the activity factor, the ClueWeb graph runs significantly faster on the GPU (by almost 8×), while the Twitter graph is slower by 3×.

Profile-driven mapping improves application performance. Given the results shown above, Luminar (i) iteratively measures the activity factor of every 20th wavefront, and (ii) updates the application's mapping to the CPU vs. the GPU based on the most recent kernel profile. To illustrate, Figure 24 shows the performance of a naive system that uses either the CPU or the GPU to run Page-Rank on all input graphs vs. the performance seen when using Luminar. Luminar is 2.4-3× faster than the naive system. These results demonstrate that a static decision about where to execute Page-Rank (CPU vs. GPU) can lead to poor performance, thus justifying the Luminar approach.

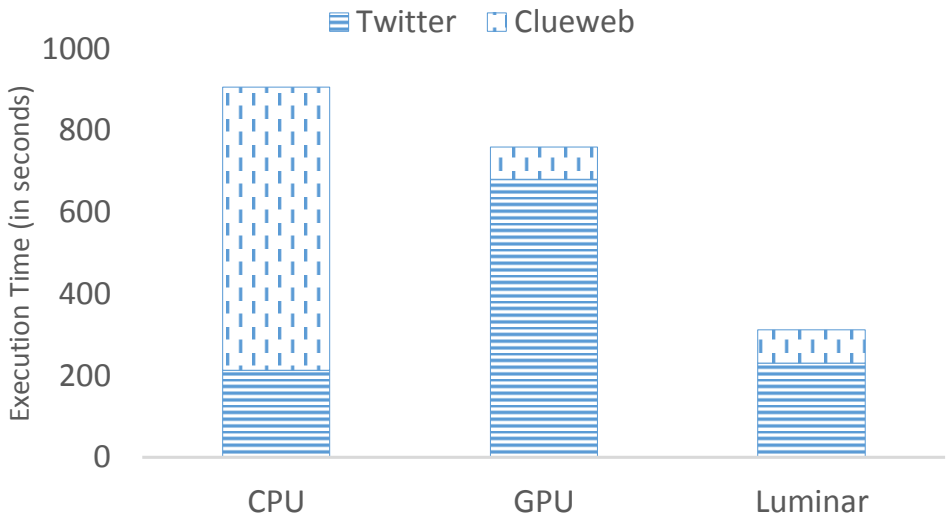

Figure 24: Page-Rank performance on a naive system using static placement vs Luminar's dynamic placement. For Luminar, the overheads associated with instrumentation and mapping are included in the overall time.

4.4.2 Runtime Code Selection

For a given algorithm, there may be multiple, alternative GPU implementations. We demonstrate how Luminar uses the activity factor metric to pick the better performing implementation of the BFS algorithm, *at runtime*, for some given input. This *Code Selector* uses the *variance* in the activity factor of the first 5 iterations of the BFS algorithm to characterize the graph as either regular or irregular. An irregular graph is one with a large skew in the distribution of edges across its nodes. Based on this characterization, the *Code Selector* then selects the better code variant to run the algorithm on the remaining iterations.

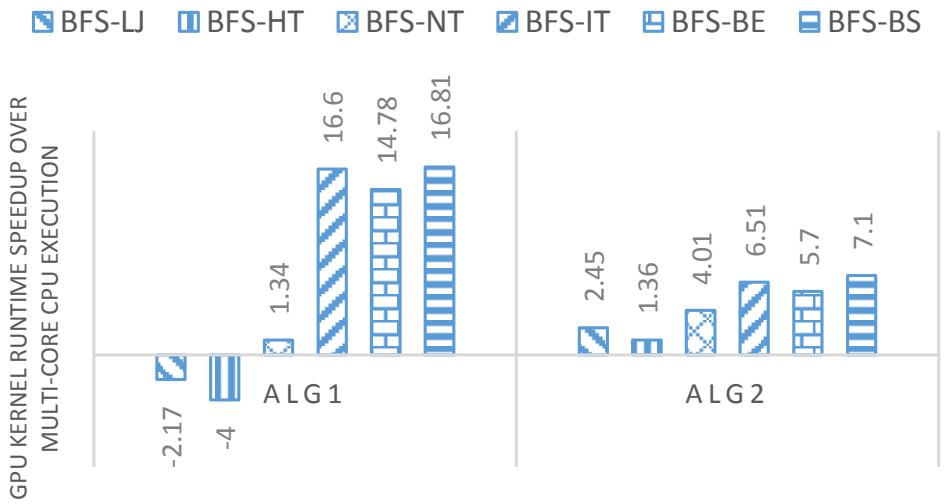

Figure 25: Speedups of GPU execution over multi-core CPU. BFS-LJ, BFS-HT, and BFS-NT are irregular graphs.

The BFS computation proceeds in a level-synchronous manner, where each iteration explores vertices at a fixed distance from the source (called frontier). There are two existing state-of-the-art GPU implementations for BFS, which we refer to as ALG1 [64] and ALG2 [43]. ALG1 uses a hierarchical queue to reduce the overheads associated with a single, global task queue for the entire GPU. This algorithm offers substantial performance improvements for regular graphs. ALG2 uses a warp-centric programming method to address workload imbalance, characterized as thread

divergence in GPU programming, resulting in better performance for irregular graphs. Figure 25 presents the speedup of the two algorithms on the GPU over multi-core CPU execution. The inputs include both regular and irregular graphs (also documented in Table 5). The plot shows that for irregular graphs ALG2 can be 3× faster than ALG1.

Activity factor can indicate irregularity in graphs. In each iteration of BFS ALG1, a GPU thread is assigned to a vertex in the frontier. And for each such vertex, the corresponding thread iterates over all its neighbors. Due to the variance in the number of neighbors, some GPU threads will be assigned more work than others. The *variance* in activity factor captures the relative skew in degree distribution, as shown in Figure 17, for the irregular `LiveJournal` and the regular `Italy` street network graphs. The same pattern is observed for all of our input graphs.

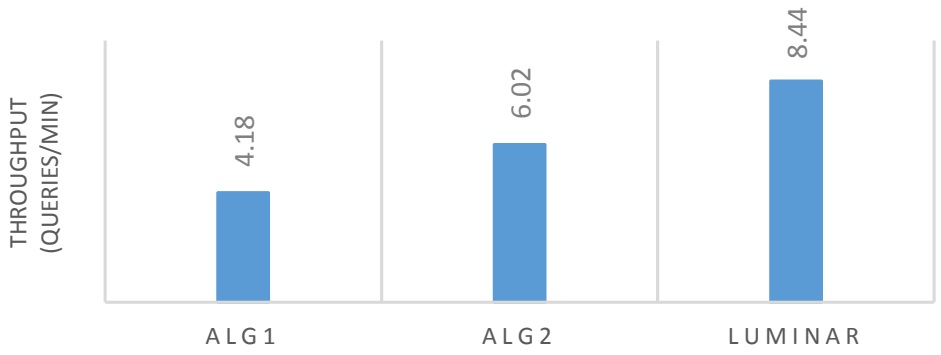

Figure 26: Throughput of BFS queries on a combination of the 6 input graphs and randomly selected source nodes.

Profile-driven selection improves performance. Luminar selectively instruments the first five iterations of the BFS query for each new input graph, and uses the activity factor metric and its variance to classify the graph as regular or irregular. This classification is used to select the corresponding algorithm implementation – the ALG2 algorithm for irregular graphs, the ALG1 algorithm for regular graphs. The approach is effective, as seen in Figure 26, which shows that Luminar's profile-driven approach improves the performance of BFS queries. We present the throughput

results of running 300 BFS queries with distinct input graphs, as described in Table 5, with random starting vertices. Further, we compare the performance of Luminar to running either ALG1 or the ALG2 implementation on all graphs. Experiments show that Luminar's approach provides the best throughput, a 40% improvement over a static choice of ALG2 and a 2× improvement over a static choice of ALG1.

4.5 A New Scheduler for Integrated GPUs

Luminar implements a novel *device affinity, contention aware* (DACA) scheduler, leveraging the profile-guided optimizations discussed earlier as well as incorporating awareness to memory contention resulting from co-locating different workloads, to improve the performance of running diverse applications on integrated GPU-CPU servers. The goal is to maximize system throughput for a set of incoming requests by leveraging GPU and CPU cores.

There are multiple challenges when scheduling applications across the CPU and the GPU. Application characteristics must be matched with the right device, and the scheduler must be able to deal with input-dependent runtime behaviors. Finally, memory contention has to be taken into account to avoid significant performance degradation when co-running different workloads on the CPU and GPU devices.

Using a combination of application runtime metrics, such as activity factor and memory intensity, and the runtime status of CPU and GPU devices, Luminar's DACA scheduler intelligently schedules applications. Luminar demonstrates how dynamic instrumentation can inform scheduling in ways that improve system throughput and energy efficiency. We use, as a baseline, online schedulers that are unaware of concepts like GPU affinity and contention: (1) the *GPU-only* scheduler simply assigns all applications to the GPU, the goal being to use the fastest device in the system, and (2) *FIFO* assigns the next task in the queue to whatever device is available, CPU or GPU. A potential third option, the *CPU-only* scheduler, assigns all applications to the multi-core CPU, but we do not discuss it further since it has the worst performance in our experiments. We

also compare DACA against the *Oracle* scheduler, which represents the best possible schedule for the given set of applications, determined statically based on device-specific execution times and observed co-run performance degradation.

The FIFO scheduler assigns the next task in the queue to an available device. Since it does not have any knowledge about the application's characteristics, the FIFO scheduler may run an application like K-Means on the CPU, thereby increasing its execution time by an order of magnitude compared to running it on the GPU. In comparison, the *device-affinity, contention-aware* (DACA) scheduler first selectively instruments applications to measure both their activity factor and memory intensity metrics, and then uses this information for subsequent scheduling decisions. Specifically, the activity factor characterization helps identify the preferred device, while the memory intensity characterization helps determine which applications should or should not be co-scheduled to avoid memory contention.

The DACA online scheduler uses a pull method, where each device has an associated thread that acquires tasks from a global task queue. Tasks are acquired as soon as the device becomes idle, using the policy explained below. The FIFO policy is obvious: it simply picks the first task on the queue. The DACA policy, in comparison, scans the queue from its beginning to find a task that has high affinity for the device, and that is likely to exhibit the least contention with the task currently running on the other device, i.e., based on its corresponding memory intensity. The DACA scheduler gives first preference to device-affinity, since the performance implications of application-device mismatch are generally more severe. The resulting policy is likely to improve system throughput: by scheduling applications on the device on which they will run best, while also avoiding co-scheduling two memory-intensive applications. Figure 27 illustrates how DACA makes scheduling decisions for a given input sequence of tasks.

DACA deals with starvation and excessive task wait times by using a *window* and an *aging threshold*. Namely, the next task to be run can be selected only from a fixed size window of tasks offset from the beginning of the queue. A large window size improves the likelihood for finding

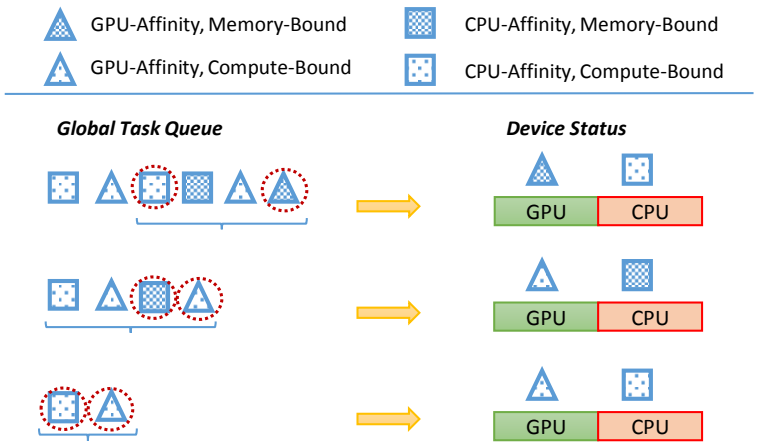

Figure 27: DACA scheduler with window size 4. Assuming similar execution time for all tasks, DACA schedules a memory-bound task with a compute-bound task in the first round, an out-of-order choice compared to FIFO. In the second and third rounds, the next two tasks in the queue are scheduled as they exhibit minimal memory contention.

two least-contending tasks, whereas a small window size limits the number of tasks executed out-of-order and experiencing excessive wait times. If a task is bypassed in the queue as many times as the aging threshold, DACA promotes it to *must run* in its selection process, ensuring it runs even if it may cause contention.

Effects of window size. An increase in window size impacts the average wait time for applications, as a greater number of tasks may be scheduled out-of-order. Figure 28 shows the scheduling penalty for tasks with latencies in the 90-100th percentile for different window sizes.

The results indicate that with a larger window size, e.g., a window size of 8, some tasks are penalized in terms of their wait times. However, larger window sizes provide the scheduler with more flexibility in choosing what applications can be co-run. Figure 29 shows the effects of window size on throughput for varying workload mixes. For compute-bound workloads, a window size of 2 is flexible enough to schedule applications based on their device-affinity, but for more memory-bound workloads, such a small windows has limited choice and thus results in poorer decisions. There are negligible differences in performance between window sizes 4 and 8. We

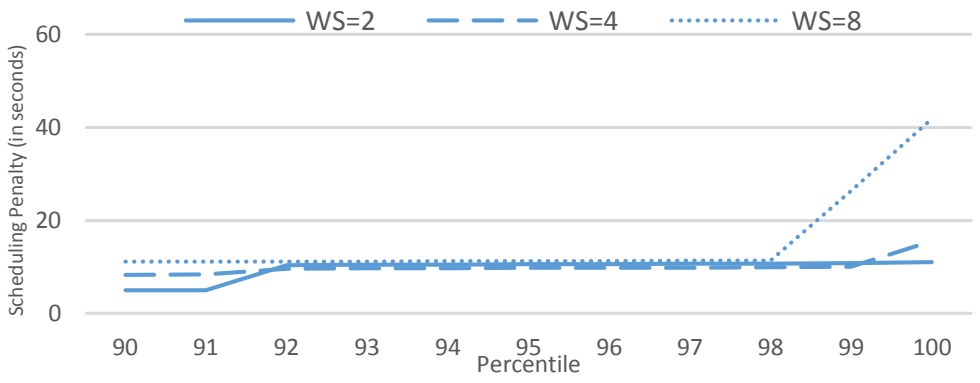

Figure 28: Scheduling penalty for tasks with latencies in the 90-100th percentile for different window sizes.

therefore set DACA's default window size and aging threshold to 4. This ensures that a task is guaranteed to run by the time two window size number of tasks are executed.

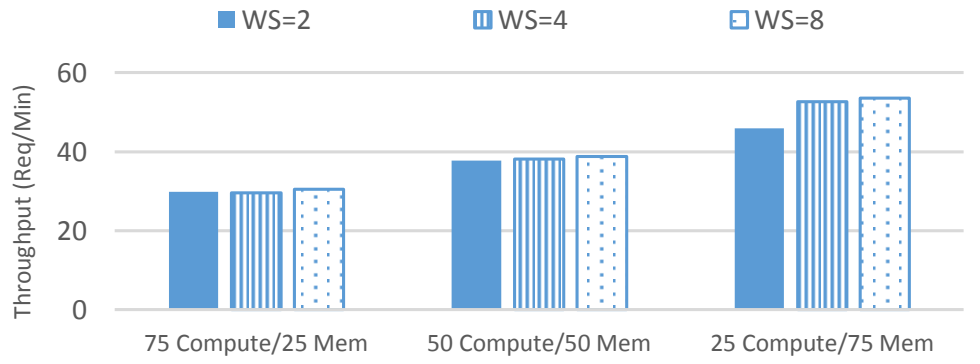

Figure 29: Throughput results for device-affinity, contention-aware scheduler for varying window sizes.

We evaluate the DACA approach on the combination of *all* workload and input pairs detailed in Table 5. We also incorporate multiple implementations of the BFS kernel to show-case runtime code selection in DACA.

Since the applications have different compute and memory characteristics, we evaluate each

scheduling policy under 600 scheduling requests with the following workload mix: (a) 75% compute-bound tasks and 25% memory-bound tasks, (b) 50% compute-bound tasks and 50% memory-bound tasks, and (c) 25% compute-bound tasks and 75% memory-bound tasks. Note that workloads exhibiting memory intensities greater than 10% are designated to be memory-bound. The workloads and datasets that constitute the 600 requests are selected at random, maintaining the workload mix criterion for each case.

Improvements in throughput. Figure 30 shows the throughput improvements achieved by DACA. We compare the DACA scheduler to the *GPU-only*, *FIFO*, *device-affinity* (DA) only scheduler, as well as to an *Oracle*. The *device-affinity* scheduler is DACA modified to not use memory intensity information, in order to assess the importance of using such information for application scheduling.

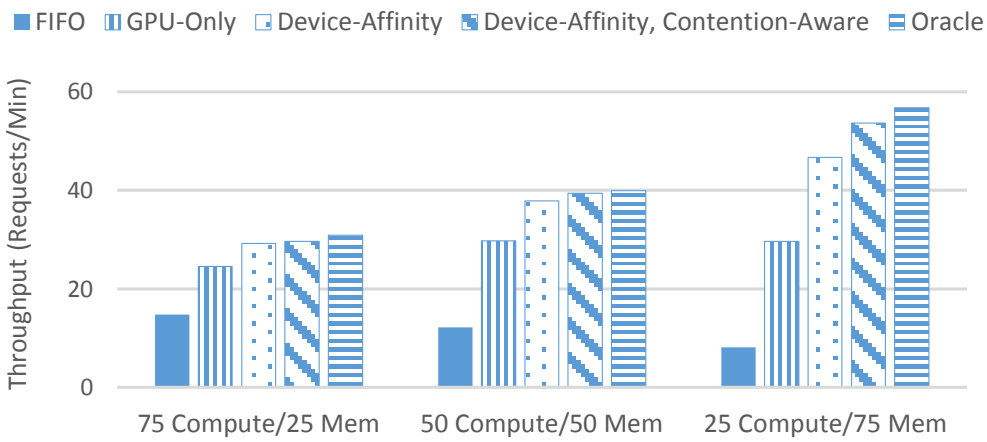

Figure 30: Comparison of scheduling algorithms on three workloads with a mix of compute and memory bound tasks.

Figure 30 shows that when 75% of the applications are memory intensive, DACA outperforms the GPU-only scheduler by 81%, and the FIFO scheduler by 6.6×. The FIFO scheduler utilizes both the CPU and GPU devices, but suffers greatly when a GPU-affinity task is scheduled on

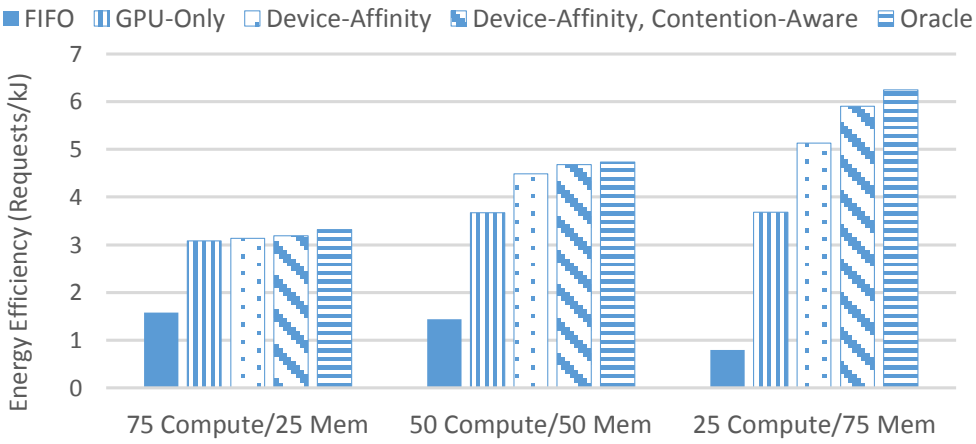

Figure 31: Energy efficiency results for DACA and the baseline schedulers.

the CPU. Compared to the device-affinity (DA) scheduler, DACA is 15% better, thereby demonstrating the importance of avoiding memory contention. However, when the workload has a high percentage of compute-bound applications, DACA provides only a 1.5% improvement, since fewer memory-bound requests implies a lower probability of contention in general. Regardless of this, using device-affinity information still improves the performance by about 2× compared to the FIFO scheduler, and 21% compared to the GPU-only scheduler. Finally, DACA's performance is within 1-6% of the performance of the Oracle scheduler.

Improvements in energy efficiency. DACA scheduling is not just important for performance, but also, to obtain the high levels of energy efficiency. We use performance-to-power ratio as a measure of energy efficiency. We measured system throughput (requests/sec) and power consumption (watt) of the APU machine to obtain throughput per watt (i.e., requests per joule). Power consumption was measured using a standard power meter.

While DACA improves energy efficiency over all baseline schedulers (Figure 31), our focus is to compare DACA's energy efficiency with the GPU-only scheduler. In general, using the GPU alone results in lower power consumption (averaging 120W versus 150W when both CPU/GPU devices are active). In spite of this, DACA achieves 3%-60% improvements in energy efficiency

over the GPU-only scheduler. DACA improves on GPU-only scheduling by efficiently using both devices, which reduces overall execution time enough to save energy in all three workload mixes.

4.6 Related Work

Irregular graph workloads. Burtscher et al. present a comprehensive workload characterization study of real-world, irregular GPU applications [23]. Nilakant et al. evaluate the performance of graph applications, such as BFS and Page-Rank, on AMD APUs [72]. Kim et al. use micro-benchmarks to characterize performance and memory contention on integrated GPUs [53]. Luminar provides online mechanisms to map computations to the best-suited device and select the optimal kernel amongst multiple implementations.

Scheduling and resource management. Several fair-share and priority-based schedulers for GPUs have been proposed in previous works, including Pegasus [42], TimeGraph [49], G-Dev [50], PTask [87] and Neon [66]. Unlike these schedulers, the goal of Luminar's DACA scheduler is to show the advantages of using dynamic instrumentation in improving overall system throughput and energy efficiency. Toward these ends, DACA leverages real-time introspection into application behavior, shown particularly important for irregular, input-dependent workloads. Scheduling policies like fair-share and priority-based methods are orthogonal to our work.

Many schedulers use relative task speedup and execution time to decide whether an application should be run on the CPU or the GPU [21, 46, 83]. In place of such coarse-grained analysis, Luminar uses fine-grained metrics to schedule and map applications. Fine-grained profiling is better suited for handling irregular, input-dependent workloads, such as graph algorithms, not addressed well by these systems. Additionally, dynamic instrumentation provides meaningful insights into application behavior which cannot be determined by execution time measurements alone, such as the degree of memory contention two candidate workloads may exhibit based on their memory intensities.

There are multiple techniques to effectively share the GPU [18, 79, 82]. Sharing is driven by

the observation that GPU devices are often under-utilized. Instead of offloading more work to the GPU, Luminar focuses on mapping each application to the most suitable resource, in ways that increase system throughput and energy-efficiency.

4.7 Chapter Summary

Luminar advocates the use of online techniques like dynamic instrumentation to improve application performance. Its *online, profile-guided* optimizations do not rely on static, offline methods for workload characterization and can automatically accelerate performance for even input-dependent, irregular workloads like graph applications. Additionally, Luminar does not require laborious code (re-)tuning when running with distinct input sets. Luminar's scheduler incorporates device affinity and contention-awareness to improve system throughput and energy efficiency, up to 80% and 60%, respectively, based on the workload mix.

CHAPTER V

LEO: A PROFILE-DRIVEN DYNAMIC OPTIMIZATION FRAMEWORK FOR GPU APPLICATIONS

5.1 Introduction

Parallel hardware, such as general-purpose GPUs, can enable high throughput in a variety of application domains, including data-intensive scientific applications [9], physical simulations [69], financial applications [80], and more recently, big-data applications [88,101]. Evidence that GPUs can improve performance over traditional CPU implementations in these domains is abundant, but manifesting such improvements for individual applications remains effort intensive, and generally requires considerable programmer expertise. Programmer-facing architectural features are a hallmark of GPU programming. Front end GPU programming frameworks support language-level abstractions to manipulate and manage specialized memories, caches, and thread geometries because exploiting the underlying architectural features is almost always required for best-case performance, and because tools that can effectively automate their use remain elusive. Consequently, optimizing GPU workloads typically requires the programmer to implement and compare multiple code versions that exercise different combinations of those features.

We argue that the current level of manual optimization effort is untenable if parallel architectures are to become more broadly applicable. GPUs have become ubiquitous in modern computing environments, resulting in more demand for generality. Performance-hungry programmers use GPUs in increasingly complex applications where algorithms rely fundamentally on unstructured or irregular control and data access patterns. GPU hardware is designed explicitly to take advantage of the regularity characterized by workloads with minimal synchronization, high arithmetic

intensity, and predictable memory access patterns. However, such pronounced regularity is not the common case for many highly data-parallel applications: graph traversal, data mining, and scientific simulations, for example, feature abundant parallelism while exhibiting data-dependent control flow and memory access patterns that are difficult to predict statically. GPU acceleration can be performance profitable for irregular data parallel workloads [23, 67, 95, 97], but typically at a significant cost in additional programmer effort. Moreover, the efficacy of code-transforming optimizations is highly data dependent for irregular codes: the need for better, automated approaches to GPU optimization is pronounced.

The recent emergence of higher-level programming front-ends for GPUs such as Dandelion [88], Copperhead [24], DSLs coded to Delite [22], and others [17, 31, 60, 81] represent an additional challenge, as well as an opportunity. Such frameworks are attractive for the degree to which they insulate the programmer from low level architectural details, yet the extent to which they can reliably and predictably exploit those features in service of performance is often limited. However, because these frameworks generate or cross-compile code to produce GPU implementations, they provide a natural interface at which a compiler and runtime can collaborate to instrument, measure, and improve generated implementations, automatically exercising code transformations commonly used in GPU optimization efforts.

We describe the design of **Leo**, a profile-driven dynamic optimization framework for GPU applications. Motivated by an emerging abundance of unstructured GPU applications that exhibit highly data-dependent memory and control-flow patterns that cannot be determined statically, Leo dynamically profiles the behavior of GPU applications using binary instrumentation, and uses the runtime characteristics of the applications to drive GPU-specific code optimizations such as memory layout transformations. The class of applications Leo targets, therefore, are streaming workloads that iteratively perform the same computations on large amounts of data, a model suitable for today's data-parallel architectures. In particular, Leo employs iterative information flow analysis

and data structure transformations to improve the memory behavior of such applications. It measures an application's runtime behavior and selectively applies optimizations during the execution of the application. Leo achieves this by integrating two existing systems: Dandelion [88] and GPU Lynx [35]. Dandelion provides the compiler framework for code transformations, and GPU Lynx provides the dynamic instrumentation framework to identify optimization strategies.

The primary contributions of Leo, therefore, are:

- An in-depth study of the memory efficiency optimization and its runtime implications on GPGPU applications.
- The design and preliminary implementation of a dynamic instrumentation and optimization framework that automatically explores code-transformation optimizations for GPUs.
- Experimental results that demonstrate the necessity, feasibility, and potential performance-profitability of such systems.

5.2 Background and Motivation

Although this chapter uses NVIDIA GPU devices and CUDA as the target platform, the same concept and technology can be applied to OpenCL supported devices.

5.2.1 GPU Metrics

The two most well-known performance limiters on the GPU are memory bandwidth utilization and thread divergence, captured via *memory efficiency* and *activity factor* [51]. These two metrics, therefore, drive the dynamic optimizations in Leo.

5.2.1.1 Memory Efficiency

Memory efficiency is a warp-level metric that characterizes the spatial locality of memory operations to global memory, the block with the highest latency in the GPU memory hierarchy. To

alleviate this latency cost, the GPU memory model enables coalescing of global memory accesses for threads of a half-warp into one or two transactions, depending on the width of the address bus. However, scatter operations, in which threads in a half-warp access memory that is not sequentially aligned, result in a separate transaction for each element requested, greatly reducing memory bandwidth utilization. Effective utilization of the GPU memory subsystem is, therefore critical to achieving good performance.

5.2.1.2 Activity Factor

Thread divergence is a common performance issue with GPU code. When threads within a warp diverge, taking different control paths, the warp serially executes each branch path taken, disabling threads that are not on that path, so non-uniform control flow entails a significant performance penalty. Activity factor (AF) characterizes how well an application utilizes the GPU SIMD parallel execution model, effectively by measuring thread divergence. Applications with completely uniform control-flow, or no thread divergence, exhibit a 100% activity factor. In contrast, applications with low AF exhibit a higher degree of control-flow irregularity.

5.2.2 A Motivating Example

In this section, we motivate the need for a profile-driven dynamic optimization framework using a concrete application, *SkyServer* [103]. The SkyServer application takes in large collections of astronomical, digital data in the form of photo objects and neighbors, and filters them to find related objects. The SkyServer workload is, in essence a series of relational equi-join operations and filtering over the two collections (see Figure 33). Differing input data distributions can yield very different selectivity for the join predicates, which in turn has a profound impact on dynamic memory access and control flow patterns in the GPU code implementing the join. We use two distinct sets of inputs to demonstrate the challenges arising from the irregular memory access patterns exhibited by this application, and discuss the solutions to this problem.

The input sets (1 and 2, detailed in Section 5.4) both work on the same total number of photo objects and neighbors, but the data distribution in set 1 yields very low selectivity for the join predicate: very few photo objects actually match neighbor objects. For set 2, the majority of the neighbor objects match the join predicate. Both photo objects and neighbor objects are defined as structures with multiple fields. A simple layout of these objects (direct mapping) generates an Array-of-Structures (AoS) data layout in GPU memory. Since each GPU hardware thread works on an individual object, the AoS layout prevents coalesced reads and writes as the members of the data structure are placed contiguously in memory, forcing different threads to access scattered memory locations. A well-known optimization to improve memory efficiency is to transform the AoS layout to a Structure-of-Arrays (SoA) layout. This results in a sequential access pattern for all threads in the same warp, improving memory efficiency. In general, the AoS-to-SoA transformation achieves significant improvements in performance on the GPU due to better utilization of the global memory bandwidth.

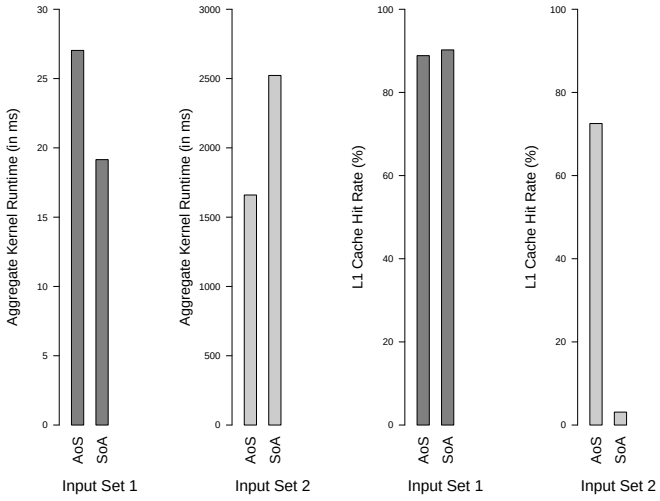

Figure 32: SkyServer runtimes and cache hit rates: (a) runtime on set 1, (b) runtime on set 2, (c) cache hit rates for set 1, and (d) cache hit rates for set 2.

This AoS-to-SoA optimization on the SkyServer application increases the memory efficiency

by a factor of two for most of its GPU kernels. However, the SoA version does not always improve the overall performance. As shown in Figure 32, while the SoA version improves the performance for the first input set, when very few objects match the join predicate, it has a negative performance impact on the second input set, when there are a large number of matches. Effective optimization for this workload needs to take into account dynamic information to deal with input-dependent performance.

The negative correlation between memory efficiency and performance for set 2 demonstrates the complexity of the GPU global memory/cache model. An SoA transformation moves members of a given object farther apart, by a factor of the array size. So when members of an object are likely to be accessed sequentially, it can lead to very high L1 cache misses when the array is large (as is the case for input set 2). Therefore, although the SoA optimization results in better global memory bandwidth utilization on the GPU, it results in poor spatial locality for members of the same object. For set 1, L1 hit rates are unaffected by the optimization because low selectivity of the join enables most intermediate data to fit in cache. However, for set 2, the L1 cache hit rate declines from 72%, for the AoS version, to only 3%, for the SoA version.

The example shows that the memory efficiency optimization does not always correlate positively with runtime performance, and its overall benefits depend on the complex interactions of GPU memory hierarchy induced by the inputs. It highlights the need for a dynamic optimization framework that not only measures an application's memory efficiency at runtime, but also evaluates the impact of a particular code transformation and makes the optimal decision at runtime. Our proposed framework, Leo, addresses precisely this need.

5.3 Design and Implementation

In this section, we present the design and our preliminary implementation of Leo. Although we envision such an auto-optimizing engine to be a part of any GPU high-level runtime infrastructure, the current design of Leo is achieved by the integration of the GPU Lynx dynamic instrumentation

library into the Dandelion compiler/runtime infrastructure.

5.3.1 System Components

As a dynamic optimization framework, Leo orchestrates the identification and selection of the optimal code and data layout transformation during the application's execution. It consists of the following two main components:

- A compilation engine that generates GPU kernel code and data layout on-the-fly from higher-level language source code.
- A JIT-based profiling engine that enables dynamic instrumentation and profiling of GPU code at runtime.

This section gives a high-level overview of these components.

5.3.1.1 Code Generation Framework

Leo leverages Dandelion to run LINQ applications on GPU. We extended Dandelion to add the necessary support to perform code and data layout transformations required by Leo. See Section 5.3.3.1 for more detail.

The Dandelion system enables the execution of Language-Integrated Query (LINQ) on GPUs. LINQ introduces a set of declarative operators, which perform transformations on .NET data collections. LINQ applications are computations formed by composing these operators. Most LINQ operators are common relational algebra operators, including projection (Select), filters (Where), grouping (GroupBy), aggregation (Aggregate) and join (Join). The Dandelion compiler automatically compiles a LINQ query into a data-flow graph and any user-defined .NET code into GPU kernels. The Dandelion runtime automatically manages the execution of the data-flow graph on GPUs and the data transfer between CPU and GPU. For example, the SkyServer application is essentially a Join followed by a filtering. Figure 33 shows the data-flow graph generated by the

Dandelion compiler. The nodes of the graph represent GPU kernels that are cross-compiled from their .NET functions.

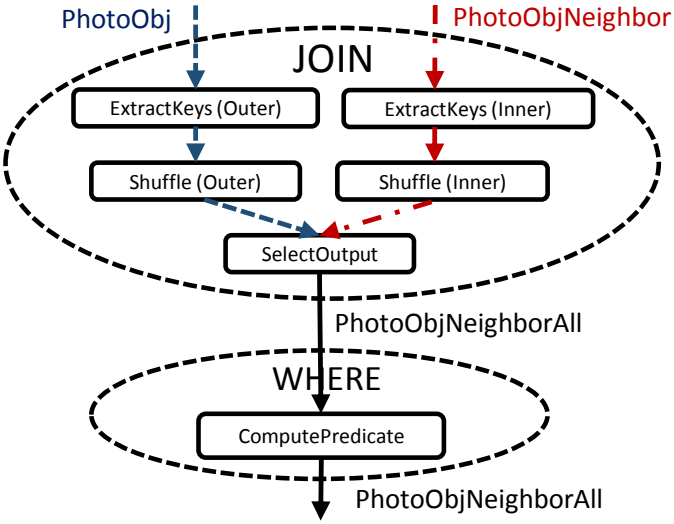

Figure 33: Simplified data-flow graph for SkyServer

5.3.1.2 Instrumentation Engine

We use GPU Lynx for dynamic profiling of GPU code. We improved Lynx significantly with a static information flow analysis [70] so that it could be used to identify the candidate data structures for optimization. See Section 5.3.3.2 for more detail.

Lynx allows the creation of customized, user-defined instrumentation routines that can be applied transparently at runtime for a variety of purposes, including performance debugging, correctness checking, and profile-driven optimizations. When built as a library, Lynx can be linked with any runtime. In the Leo framework, Lynx is integrated with the Dandelion compiler/runtime to support the execution of CUDA kernels representing the LINQ relational algebra primitives, on NVIDIA GPU devices. Lynx provides a parser and intermediate representation (IR) abstraction for extracting and generating NVIDIA's Parallel Thread eXecution (PTX) from the compiled CUDA fat binary. An essential feature of GPU Lynx is its flexibility and extensibility, enabling

both the specification of user-defined instrumentations using its C-based API, and the creation of sophisticated control-flow and data-flow analyses from its IR abstraction.

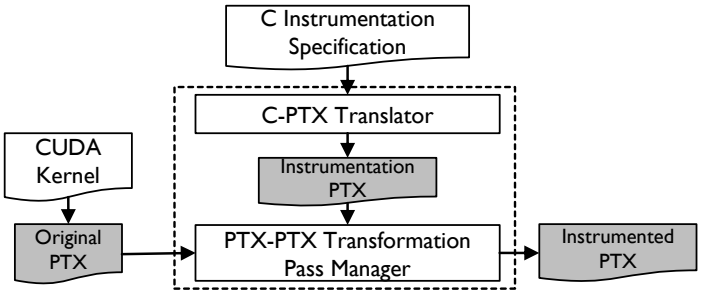

Figure 34: GPU Lynx Instrumentation Engine

An overview of the GPU Lynx instrumentation engine is shown in Figure 34. A C-based instrumentation is provided to the framework in addition to the original GPU kernel. The specification defines where and what to instrument. Lynx allows instrumentations to be defined at the kernel level, basic block level, or the instruction level. The Lynx engine generates the final instrumented PTX kernel from the C specification and the original PTX kernel, by enlisting the C-PTX Translator and the PTX-PTX Transformation Pass Manager.

5.3.2 System Overview

The Leo runtime orchestrates the identification and selection of the optimal code transformations and data layouts for GPU kernels. The computation model we support is based on streaming, i.e., the input is divided into chunks and chunks are transferred to GPU concurrently with the GPU execution. This model enables Leo to make optimization decisions based on the execution of preceding chunks. In the current design, Leo runs the Lynx instrumented code for the first chunk to determine possible candidate kernels for optimization. This allows Leo to generate the optimized version of the code with the necessary code and data layout transformations. We then run the second and third chunks with and without the optimizations respectively, and compare the total elapsed running times to determine which version of the code to use for the subsequent

chunks. This profiling is repeated at continuous intervals to detect time-varying runtime behaviors and relevant application phase changes.

Figure 35 presents a high-level overview of the design of the Leo framework, depicting the general steps the runtime takes in order to apply profile-driven optimizations to LINQ applications. We describe these steps more concretely in the context of the *SkyServer* application introduced in Section 5.2.2.

In Step 1, we use Dandelion to generate the original version of the *SkyServer* GPU code, which results in an AoS data layout in memory. In Step 2, we apply Lynx to generate an instrumented version of the GPU code. For the data layout transformation, the code is instrumented with the memory efficiency metric to characterize the spatial locality of global memory accesses. In Steps 3 and 4, the instrumented code is executed and the profiling information is collected. The original GPU kernels in the *SkyServer* application exhibit a low average memory efficiency (less than 40%, as shown in Figure 39(a)). The instrumented results capture global memory load and store accesses for all possible data sources in the code. As one might expect, not all data sources may necessarily exhibit poor memory efficiency. As a result, Leo applies information flow analysis to link each global memory load/store access in the GPU kernel code to its corresponding data source. This enables Leo to precisely identify the data structures that need to be transformed. The AoS-SoA code and data layout transformations are applied to the candidate data structures in Step 5 to generate an optimized version of the GPU code. However, as discussed in Section 5.2.2, the *SkyServer* workload exhibits input-dependent performance with the AoS-SoA optimization. The optimization is effective when the join predicate has low selectivity (input set 1) but degrades performance when the join predicate has high selectivity (input set 2) due to the L1 cache effects. To deal with this input-dependent behavior, Leo uses the execution of the second and third chunks, with and without the optimization, respectively, to select the optimal data layout for all subsequent chunks.

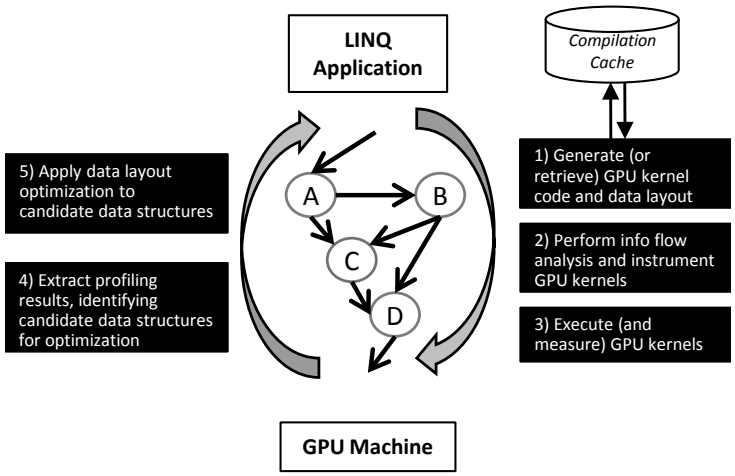

Figure 35: High-level overview of Leo.

5.3.3 Implementation Details

We now provide a more detailed description of the new and important features of Leo.

As described before, the Dandelion compiler compiles a LINQ application to a data-flow graph, where the nodes represent fragments of the computation and the edges represent communication channels. For execution on the GPU, each node translates to a primitive relational algebra operation, such as a *Select*, *Join*, or *GroupBy*. The generated CUDA primitives are generic: the input and output data types, as well as the user-defined functions are template parameters of the primitives. The Dandelion compiler instantiates the primitives using the generated GPU data-types and code.

5.3.3.1 Code and Data Layout Transformation

In addition to code generation, Dandelion also handles the runtime management of data buffers across input/output (I/O) channels, seamlessly allocating buffers on-the-fly, without programmer intervention. The layout of a particular data structure, therefore, can be modified by the Dandelion compiler dynamically, across the various I/O channels. This implies that a given node is capable of receiving data in one form but the subsequent node can receive the same data in another form.

Listing 5.1: Original (AoS) Version for SkyServer Example Function

```
__device__ Compute(PhotoObj& p, PhotoObjNeighbor& n){
  struct PhotoObjNeighborAll pn;
  pn.p = p;
  pn.n = n;
  return pn;
}
```

Listing 5.2: Optimized (SoA) Version for SkyServer Example Function

```
__device__ Compute(PhotoObj*p,
PhotoObjNeighbor*n, int pLen, int pIdx, int nLen, int nIdx){
struct PhotoObjNeighborAll pn;
int s = 0;
pn.p.a= *((long*)(p+s*pLen+sizeof(long)*pIdx);
s += sizeof(long);
pn.p.b= *((int*)(p+s*pLen+sizeof(int)*pIdx);
s += sizeof(int);
pn.p.c= *((int*)(p+s*pLen*sizeof(int)*pIdx);
//...
s = 0;
pn.n.a= *((long*)(n+s*nLen+sizeof(long)*nIdx);
s += sizeof(long);
pn.n.b= *((long*)(n+s*nLen+sizeof(long)*nIdx);
//...
return pn;
}
```

```
for basic block bb in reversed cfg do
    for instruction inst in reversed bb do
        if inst = global mem load or store then
            registerSet s = {address operand of inst}
            flowMap[inst] = s
        end
        else
            for [instruction m, registerSet s] in flowMap do
                for register r in s do
                    if dst reg operand of inst = r then
                        if inst = param mem load then
                            flowMap[m] = flowMap[m] + (src operand of inst) - r
                        end
                        else
                            flowMap[m] = flowMap[m] U ({src reg operands of inst} - r)
                        end
                    end
                end
            end
        end
    end
end
```

Figure 36: Information Flow Analysis Algorithm

All generated CUDA kernels are stored in the compilation cache to avoid runtime JIT compilation overheads for subsequent data chunks and application runs.

For the data structures identified for optimization, Leo performs an AoS-SoA data layout transformation on these candidates. The transformation involves modifications to both the generated CUDA code and the data layout. Leo invokes the Dandelion compiler to generate the SoA versions of the CUDA code and switches from the row-major to the column-major data layout for the identified data structures.

Listings 5.1 and 5.2 show the original (AoS) and optimized (SoA) versions of an example *auto-generated* function, `Compute`, in the *SkyServer* application. The `Compute` function performs the

necessary data accesses for the photo object and photo object neighbor structures, and is invoked by the `SelectOutput` kernel (shown in Figure 37). In the original `Compute` function, an AoS memory layout is assumed, allowing the code to directly access the individual members of the two structures. The transformed version of this function assumes an SoA memory layout, requiring additional parameters to index the individual members of the structures appropriately in column-major order. Since the `Compute` function is automatically generated at runtime by the Dandelion compiler, we have the flexibility to modify the function prototype as needed.

5.3.3.2 Information Flow Analysis and Dynamic Instrumentation

Leo uses a combination of information flow analysis and dynamic instrumentation to identify data structures that exhibit irregular global memory access patterns at runtime. The goal of information flow analysis is to link each individual global memory load or store instruction in the GPU kernel code to its corresponding data source. Since not all data structures may need to be transformed, it is important to precisely know which data structures are candidates for optimization. This mechanism enables just that.

Leo uses a form of information flow analysis, known as taint analysis [70], to track causal dependencies between global memory operations, and kernel parameters that identify the data sources for those memory operations. In this analysis, all addressable variables in global memory instructions, and corresponding registers that participate in the calculation for those addressable variables, are marked as tainted. Using a backward data-flow analysis, each addressable variable's data source is tracked back to its corresponding kernel parameter. In the CUDA programming model, all kernel parameters are stored in the GPU parameter memory. The parameter must be loaded into a register from the parameter memory before being used. This information is used to determine each global memory operation's data source.

The information flow analysis algorithm is presented in Figure 36, and a visual depiction for an example GPU kernel, the `SelectOutput` kernel from the SkyServer application, is shown in

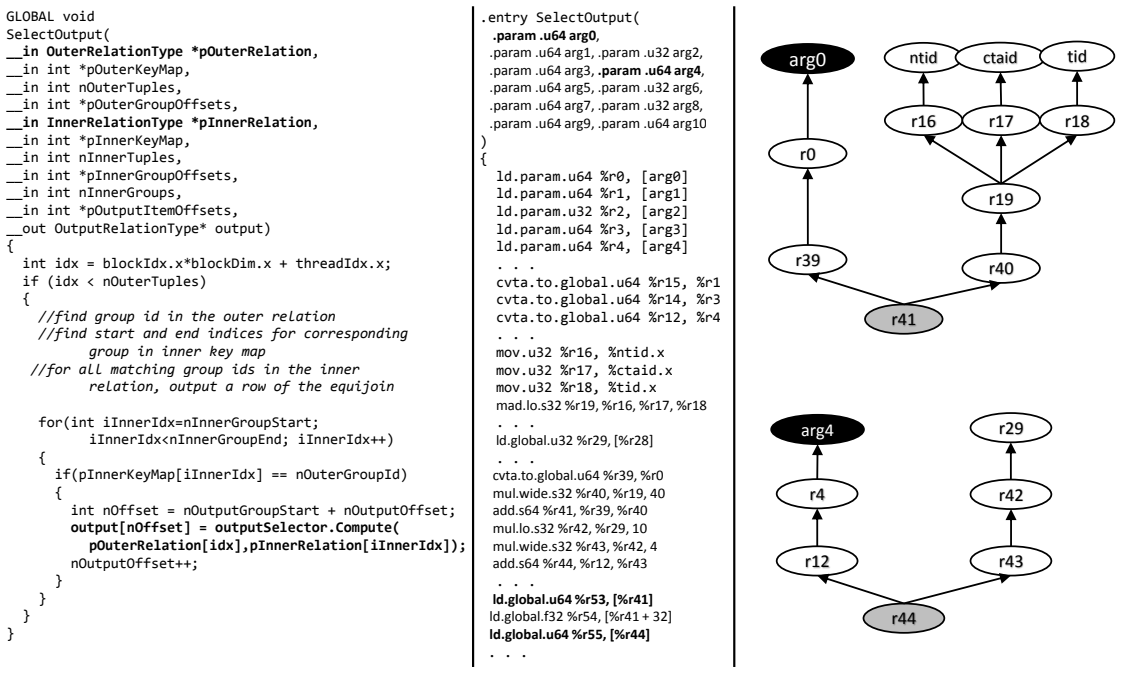

Figure 37: Information flow analysis: (a) CUDA code for the *SelectOutput* kernel, (b) PTX snippet of the same kernel code, and (c) visual depiction of the information flow analysis algorithm, showing the mapping of two global memory addressable variables to their respective data sources, identified as kernel input parameters.

Figure 37. Specifically, Figure 37 shows how global memory addressable variables *r41* and *r44* are linked back to their corresponding kernel input parameters, *arg0* and *arg4*. Note that *arg0* maps to the outer relation (*pOuterRelation*) and *arg4* maps to the inner relation (*pInnerRelation*) of the cross product join operation in the original `SelectOutput` CUDA kernel.

A mapping for each global memory operation to its corresponding input data source is constructed. Each global memory operation is identified by an index that determines its position in the static PTX kernel code. This index is returned by GPU Lynx's instrumentation API function, *memOpId()*. For the `SelectOutput` kernel code snippet in Figure 37, the mapping would look like the following:

```
memOpId: 0  --> ....
memOpId: 1  --> pOuterRelation
memOpId: 2  --> pInnerRelation
```

Once such a mapping is established, the kernel is instrumented with the memory efficiency metric. The memory efficiency instrumentation is defined as follows. For every global load or store instruction, each thread within a thread block computes the base memory address and stores it in shared memory. For NVIDIA GPUs, a half-warp, or 16 threads, can coordinate global memory accesses into a single transaction. This implies that if the base address is the same for all threads belonging to a half-warp, then the memory accesses will be coalesced. A single active thread within a warp, the one with the smallest index, is selected to perform an online reduction of the base addresses written to the shared buffer. It maintains a count of unique base addresses within a half-warp, to determine the total number of memory transactions required for a particular memory operation. The number of transactions required for a particular memory operation by a given half-warp is stored in a global memory instrumentation buffer, indexed by the memory operation index. After the kernel completes execution, the memory operation index is used to determine the individual memory efficiency for each kernel parameter. The C-based instrumentation specification

for the memory efficiency metric is shown in Listing 5.3.

Listing 5.3: C Specification for Memory Efficiency Metric

```
ulong threadId = blockThreadId();
ulong warpId = (blockId() * blockDim() + threadId) >> 5;
ON_INSTRUCTION:
MEM_READ:


MEM_WRITE:
GLOBAL:
{
  sharedMem[threadId] = computeBaseAddress();
  if(leastActiveThreadInWarp())


  {
    uint offset = (memOps() * warpId + memOpId())*2;
    globalMem[offset] += uniqueElementCount(sharedMem);
    globalMem[offset+1] += 1;
  }


}
```

5.3.3.3 Extraction of Profiling Data

In order to determine if a particular code transformation is beneficial, the kernel runtimes of the generated original and optimized CUDA kernels are measured via GPU Lynx's kernel runtime instrumentation [35]. The kernel runtime instrumentation polls hardware counters on the GPU device, exposed as PTX instructions, which capture precise measurements of multiple events within

the execution of a single kernel without including latencies of PCI bus, driver stack, and system memory.

GPU Lynx exposes an API to Dandelion for retrieving profiling results. The profiling results are stored in a multi-dimensional map (JSON), identified by each kernel's name at the first level, and the profiling metric at the second level. The profiling results include the necessary meta-data to identify the candidate input data structures for optimization as well. After the execution of the instrumented kernel code, the profiling results of the run are retrieved by the Dandelion compiler. The Dandelion compiler uses this information to selectively apply code transformations to the candidate data structures.

The decision model to select appropriate data structure candidates for optimization can incorporate various pieces of information, such as a threshold on the memory efficiencies of the data structures, the size of the data structures, and so on. For instance, all data structures that have a memory efficiency lower than a specified value may be selected as candidates for optimization. In the current implementation, we try to aggressively optimize all data structures that have a memory efficiency lower than 100%, and input sizes greater than the L1 cache size.

5.4 Evaluation

We evaluate our design using a preliminary prototype of Leo using four applications from different domains: Black-Scholes, K-Means, SkyServer [103], and Bellman-Ford's Single-Source-Shortest Path algorithm. The current Leo prototype is not a full end-to-end implementation: while it integrates dynamic instrumentation and measurement for activity factor and memory efficiency, along the necessary support at the generic primitive library targeted by the Dandelion compiler, manual intervention is still required to code alternate versions of anonymous functions in the cross compilation from C# to CUDA. While incomplete, we hope that our preliminary results offer useful insights on the potential profitability, feasibility and necessity of a dynamically adaptive, auto-optimizing GPU framework.

All benchmarks are coded using .NET LINQ as described in Section 5.3.1.1. As a result, the cross-compiles GPU code for each benchmark relies on a number of separate kernels. We evaluate our profile-guided optimizations by considering their impact on the performance of the the most compute-intensive (and consequently, longest-running) of those kernels in isolation, and by considering their impact on end-to-end performance for a subset of the benchmarks: Black-Scholes, K-Means, and SkyServer. Details of the evaluation platform and benchmarks are provided in Tables 6 and 7 respectively.

Table 6: System configuration for Leo's experimental evaluation

CPU	Intel Xeon E5504 @ 2.00 GHz
GPU	Tesla M2075, 448 CUDA cores
Operating System	Windows Server 2008 (SP1)
CUDA Version	5.5

Table 7: Selected workloads for Leo

Application	**Description**	**Input Configurations**
Black-Scholes	N options	N=1x10^7
K-Means	M N-dim points, k clusters	M=1x10^6, N=32, k=40
SkyServer	O objects, N neighbors	Input 1: O=2048, N=2x10^7, up to 200 matching neighbors Input 2: O=2048, N=2x10^7, up to 2x10^7 matching neighbors
Single-Source-Shortest-Path	*N* nodes, *M* edges	USA road network: N=24x10^6 , M=58x10^6

5.4.1 Black-Scholes

The Black-Scholes algorithm estimates option prices based on a number of parameters including constant price variation, strike price, time-to-expiry, etc: coded in LINQ, the workload is a single `Select` statement using a lambda function to encode the algorithm. Changing the Black-Scholes data layout from an AoS to an SoA representation, we improve the memory efficiency from **18%**

to **100%**. The workload is largely compute-bound: a low ratio of memory accesses to compute operations limits the corresponding kernel speedup to just $1.12\times$.

5.4.2 K-Means

K-Means is a classical clustering algorithm which partitions M N-dimensional points (or vectors) into k clusters by repeatedly mapping each point to its nearest center, and then recomputing the cluster centers by averaging the points mapped to each center. The workload is coded to LINQ as a `GroupBy` followed by a `Select`, the former of which uses a method called `NearestCenter` as the key extractor function. The cross-compiled Dandelion code will rely on a number of lower-level primitives to implement the relational algebra, but we focus on the corresponding `NearestCenter` GPU kernel as its execution overwhelmingly dominates end-to-end performance for the workload.

The primary data structure is a collection of N-dimensional points, whose most obvious in-memory representation arranges the dimensions of each point in contiguous memory locations. If the number of dimensions is large, such a layout yields poor memory efficiency on the GPU. Figure 38(a) shows the memory efficiency for the original and the AoS-SoA transformed versions for this code, with varying dimensions, and Figure 38(b) shows the corresponding kernel speedup. With $N = 1$, the layouts and memory efficiency are predictably equivalent. As N increases, the original version's memory efficiency degrades, while the optimized version's memory efficiency remains high.[1] Memory efficiency has a first-order impact on performance for this workload as N increases, providing $27\times$ improvement at $N = 32$.

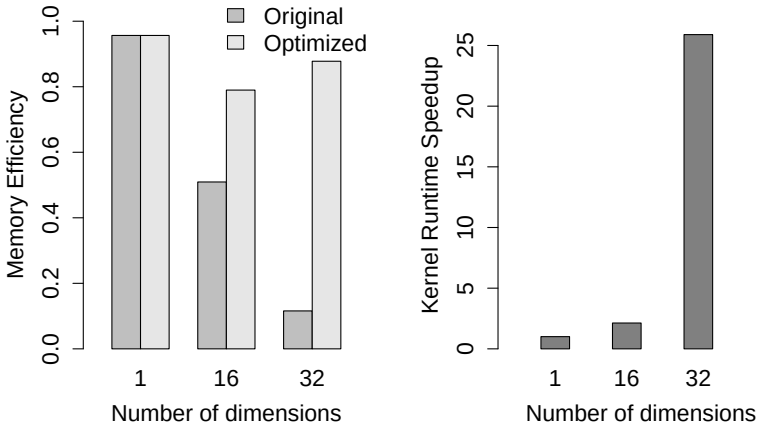

Figure 38: K-Means (a) memory efficiency and (b) kernel runtime speedup for varying dimensions. As the number of dimensions grows, the memory efficiency of the original AoS code version degrades while the optimized SoA version's remains high. Memory efficiency has a direct correlation on kernel runtime performance for this workload.

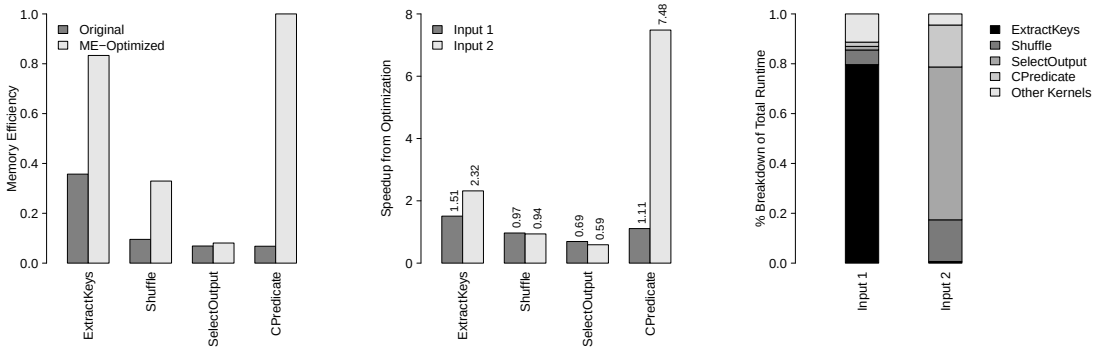

Figure 39: SkyServer (a) memory efficiency of original and optimized kernels, (b) individual kernel runtime speedups and (c) computation breakdown of all the kernels for the two distinct input sets.

5.4.3 SkyServer

SkyServer takes as input collections of digitized astrological images (encoded as "photo objects") and the relative locations of images (encoded as "photo neighbors"). The workload filters these data according to criterion that enables the identification of related astrological objects. It is expressed in LINQ as a series of `Join` operations over the objects and neighbors collections.

Dandelion implements the underlying relational algebra on GPUs using techniques fundamentally similar to hash-join [32], decomposed into a number of GPU kernels. The approach first identifies items in the input relations matching the join predicate, then shuffles matching items per-relation into contiguous positions, then computing the final join output as the cross-product of items in each (matching) contiguous block. While a deep understanding of the implementation is not required to here (we refer the interested reader to [88]), some details play an important role in the profitability of the optimizations performed by Leo: to first order, four kernels corresponding to the steps above dominate the performance of SkyServer end-to-end performance, called `ExtractKeys`, `Shuffle`, `SelectOutput`, and `ComputePredicate`, so our evaluation effort focuses on Leo's ability to reduce compute latency for these four kernels.

We evaluate SkyServer with two input sets (1 and 2), corresponding to different levels of selectivity for the join predicate. In set 1, very few of the neighbors (up to 200) match the predicate against photo objects, and vice versa for set 2, where almost all of the photo neighbors are matches. Consequently, for set 1, the `SelectOutput` kernel is the least significant contributor to the overall computation, but forms the largest component for set 2.

Figure 39 presents (a) memory efficiencies of the original and optimized kernels, (b) individual kernel speedups achieved by the AoS-SoA optimization, and (c) the computation breakdown of the kernels for the two input sets. The transformation improves memory efficiency for all of

[1]The K-Means `NearestCenter` kernel takes two vectors as input: one for the points and one for the cluster centers. The number of centers is generally much smaller than the number of points, and fits in L1 cache for all input sizes we consider, so our optimization focuses on the points collection.

SkyServer's kernels, but the improvement is not as significant for `SelectOutput`. Although the transformation ensures that the loading of members of the SkyServer's input data structures are coalesced, sequentially indexed threads may not necessarily be accessing contiguously located elements in the array due to the data skew introduced by the hash function and the join predicate.

`ExtractKeys` and `ComputePredicate` benefit from the AoS-SoA transformation for both input sets, whereas `Shuffle` and `SelectOutput` are negatively impacted in both cases. This is due to the tension between memory coalescing and cache performance discussed in Section 5.2.2. The data structures used in SkyServer comprise 10 long integer and floating point fields, so high selectivity of the join predicate (many matches) results in high L1 miss rates under the SoA layout for `Shuffle` and `SelectOutput` as they must collect data spread across many cache lines to shuffle individual records into a logically contiguous arrangement. When the join predicate has low selectivity, `Shuffle` and `SelectOutput` constitute only about 7% of the entire computation, so the negative impact of the transformation on those kernels is masked by the benefit enjoyed by `ExtractKeys` and `ComputePredicate`. We conclude that the profitability of this optimization is input-dependent for SkyServer, highlighting the need for a dynamic framework to select the best code transformations at runtime.

5.4.3.1 Exploring Activity Factor

Recall from Section 5.2.1.2 that Activity factor (AF) essentially characterizes the level of thread divergence. In future work, we plan to incorporate AF to drive dynamic optimizations. As step towards that goal, we use SkyServer to demonstrate that AF can be a useful predictor for determining hash table sizes on-the-fly.[2]

Recall that for the *join* operation, keys are inserted into a lock-free hash table using a compare-and-swap (CAS) operation. As a result, the distribution of the keys, the hash function, and the size

[2]Many relational algebra primitives in Dandelion such as *Join* and *GroupBy* rely on efficient hash table performance.

of the bucket list array can yield variance in the length of the bucket lists, which in turn causes thread divergence as threads traverse different length lists concurrently. Smaller bucket list arrays exacerbate this effect by forcing longer bucket lists, while over-provisioning the bucket list array wastes memory, which is particularly scarce in big-data workloads such SkyServer. Dynamic resize is an unattractive option in this context primarily because it is an inherently sequential operation whose additional can be worse for performance than the thread divergence incurred by tolerating a poorly sized table. Consequently, hash-table bucket array size is a parameter managed by the Dandelion compiler. Because many applications have performance-sensitivity to the hash-table size, Dandelion currently provides an interface for programmer hints for this parameter, but even when values are well-chosen the situation is not optimal, as the hash-table size remains constant even if the application's needs change dynamically.

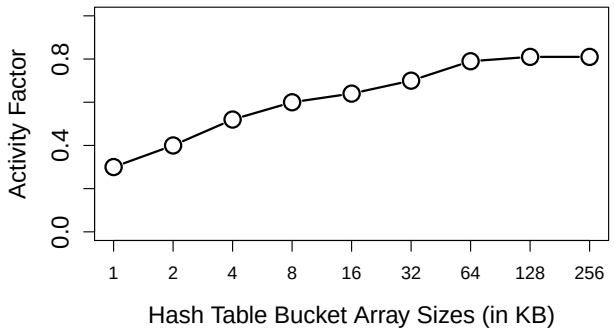

Figure 40: SkyServer configured with input set 2: activity factor for varying hash table bucket array sizes.

To demonstrate how activity factor can guide dynamic selection of the hash table size we measure the AF in SkyServer for various hash table sizes, shown in Figure 40. The data show that AF improves as the hash table size increases up to a 64KB, leveling out as sizes increase beyond that, suggesting an good choice of hash table size for the given data distribution. The same mechanisms used to communicate and respond to ME in the runtime can be used to select better

hash table sizes based on AF.

5.4.4 Single-Source Shortest Path

Single-Source Shortest Path (SSSP), is Bellman-Ford's algorithm for finding the shortest path from a source node to every other node in a graph. The algorithm is based on the principle of relaxation, in which iteratively improves an approximation until converging on an optimum. The data structures in our implementation are nodes (an id and distance) and edges (source, destination, weight).

Figure 41 shows the kernel runtimes and L1 hit rates for five iterations of this algorithm on the USA road network graph, for both the original and memory-efficiency optimized code versions. In aggregate, the AoS-SoA transformation improves memory efficiency from **71%** to **100%**, but the data show significant time-varying behavior. In early iterations, the SoA layout version performs better because the join predicate has low selectivity when only a few nodes have been visited. Over time more and more nodes are visited, increasing selectivity which results in the join spending more time shuffling objects in the correct regions. Since the SoA layout spreads individual object members across different cache lines, this in turn causes higher L1 miss rates, and in later iterations, the original layout is more performant by larger and larger margins. This time-variance can be handled by our periodic re-evaluation scheme, which enables the runtime to detect and respond to such trends in application behavior.

5.4.5 End-to-End Performance

In this section, we consider the end-to-end impact of memory efficiency optimizations on the performance of a subset of benchmarks: Black-Scholes, K-Means, and SkyServer.[3] The goal of this

[3]We do not include end-to-end data for SSSP because the benchmark uses the `Concat` LINQ operation. Because Dandelion currently lacks a GPU-side implementation for this operation, data must be moved back and forth to the CPU for that phase of the computation at a performance cost that dominates end-to-end performance. Until GPU-side `Concat` support is available, performance comparisons at the GPU kernel level are meaningful, but end-to-end performance measurements cannot provide an accurate picture.

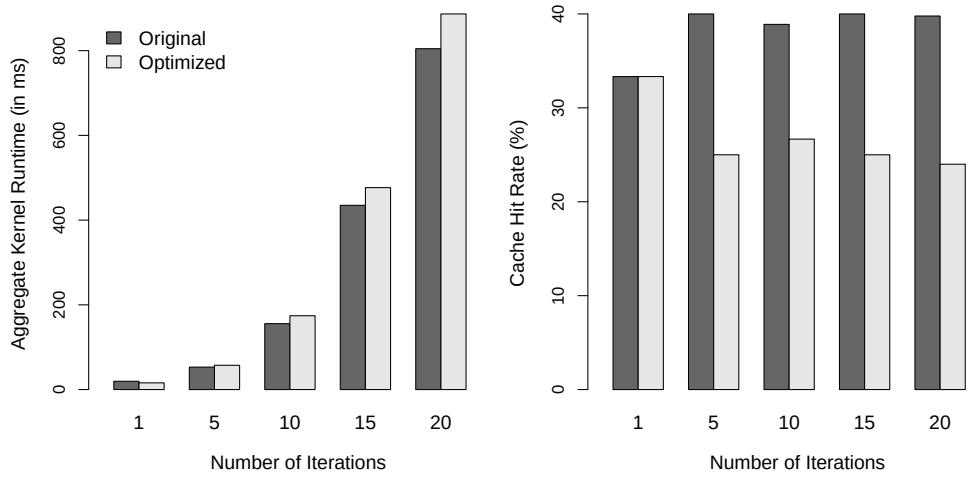

Figure 41: Single-Source-Shortest-Path (a) aggregate kernel runtimes and (b) cache hit rates for five distinct iterations on the same input (USA road network graph)

experiment is to demonstrate that Leo is able to amortize the runtime overheads incurred via dynamic instrumentation over a relatively small number of iterations and still provide significant performance gains in some cases. In the first iteration, we instrument GPU code and measure its memory efficiency, using that to guide the decision about whether to generate a different optimized version. The second and third iterations are used to measure performance for optimized and unoptimized versions (without instrumentation), enabling the runtime to select the better of the two, which is used for the remainder of the iterations.

Table 8: End-to-end performance of selected applications with Leo's profile-guided optimizations

Application	**Profile-Guided**	**Oracle**
Black-Scholes	**1.07**	1.09
K-Means	**1.09**	1.10
SkyServer Input 1	**1.34**	1.52
SkyServer Input 2	**1.42**	1.53

Table 8 compares the speedup over unoptimized code of profile-guided optimization against the speedup that would be attained if a perfect oracle in the compiler could select the most performant code version with no overhead. The code version represented by the oracle is the hand-optimized,

statically performed AoS-SoA transformation applied to kernels that benefit from this optimization. The *Profile-Guided* column presents the speedup that we observed from using our compiler framework, which must amortize overheads of instrumentation, measurement, optimization application, and reloading the GPU device when code versions are changed. The data assume that the compilation cache hides the overheads of JIT compilation, so compile overheads are not included. The *Oracle* column shows the speedup attained if the most optimal code version can be selected at the first iteration with no additional overhead.

For Black-Scholes and K-Means, the potential benefit is modest (7%-9%), the profile-guided runs achieve speedups very close to those obtained with an oracle. In contrast, for SkyServer the potential benefit is significant. The SkyServer application has the most complex data structures and generated code, compared to the more simpler implementation of the other two workloads. As a result, the instrumentation overheads are more significant as well. We note that the difference in maximum profitability of optimization across these workloads is expected. Black-Scholes and K-Means are compute-bound, so their lower memory intensity relative to SkyServer translates to fewer sites at which instrumentation code is inserted and a correspondingly less runtime overhead relative to SkyServer.

The data also show that as one would expect, additional overheads in the end-to-end scenario (notably CPU-GPU data transfers) attenuate the speedups observed in when GPU kernels are measured in isolation. Emerging integrated CPU-GPU architectures, programming abstractions and runtime tools that maximize asynchrony and/or eliminate unnecessary data migration can lessen this impact, enabling our framework to deliver higher gains in future systems. In the future, we will explore application of finer-grained instrumentation techniques to alleviate instrumentation overheads for memory-bound codes, such as selectively instrumenting only certain kernel segments or certain types of memory operations. This may represent a profitable avenue for lowering overheads if sampling alone is sufficiently predictive.

5.5 Related Work

Profile-Guided Optimization. Adaptive, dynamic, and profile-guided optimization techniques have enjoyed much research attention over the past few decades [11]. Leo draws from basic techniques described in the literature, and we claim no contribution in this domain other than synthesizing known techniques in a new context.

Higher-Level Language Front-ends. The research community has focused considerable effort on higher-level programming front-ends for GPUs [22,24,88], with the primary goal of insulating the programmer from complexities induced by the architecture. The Delite compiler and runtime framework [22] performs domain-specific optimizations on DSL applications for execution on multiple heterogeneous back ends. Delite shares many common features with our framework: support for programmer productivity without sacrificing performance, representation of applications as execution graphs to enable runtime scheduling and optimizations, and heterogeneous code generation for both CPUs and GPUs. The optimizations explored by the Delite runtime focus on fusion of Delite operators to reduce memory pressure and improve cache behavior, which result in fewer memory allocations and total number of accesses, as well as runtime scheduling decisions. Delite's extensibility enables compiler optimizations that are aware of the semantics of operations within the domain, while Leo searches for low-level input-dependent optimization opportunities that are inaccessible to an optimizer with a static view, however semantically rich that view may be. Copperhead [24] is a high-level data parallel language embedded in Python, enabling the execution of Python applications on parallel architectures. Leo's novelty lies in its integration of a dynamic instrumentation engine, GPU Lynx, with a cross-compilation runtime, Dandelion, to enable profile-driven optimizations, transparently and seamlessly, based on the application's runtime behavior.

GPU Optimizations. The GPU-specific optimizations, such as the AoS-SoA transformation, have been studied extensively in previous works as well [86, 98, 100, 104]. In [86], Rompf et al.

show how the AoS-SoA data structure optimization can be performed via internal compiler passes. DL [98] is an OpenCL-based runtime library that provides an efficient data layout transformation engine, specifically to perform AoS-SoA type transformations. G-Streamline [104] is a framework which removes dynamic irregularities in GPU applications on-the-fly, such as those resulting from irregular memory accesses and data-dependent control accesses. Leo's goal is to automatically and transparently determine when a particular optimization is useful or not, and respond to the given application's varying runtime behaviors dynamically. As such, libraries such as DL and G-Streamline are complementary to our work, and can be linked with our framework to provide the data layout re-ordering mechanisms for optimizing irregular memory and control-flow accesses.

Irregular Workloads. The ubiquity of irregular, unstructured applications running on GPUs has made the need for an auto-optimizing framework that reacts to the application's runtime behavior increasingly urgent. In general, GPU acceleration for irregular data parallel workloads [23, 67, 95, 97], has been studied extensively, and augments the potential value of our proposed framework for current and future heterogeneous systems.

5.6 Chapter Summary

The increasing ubiquity and attractive performance properties of parallel architectures has resulted in increased demand for generality and ease of programming, and driven the emergence of higher-level front-end programming languages and targeting GPUs. While such tools can insulate the programmer from complexity and low-level architectural detail, that insulation shifts the responsibility of efficiently exercising the architecture from the programmer to compilers and runtimes, in some cases sacrificing the goal of achieving best-case performance. Leo is a dynamic optimization framework whose goal is make such sacrifice unnecessary by enabling the system to automatically search the implementation and optimization space formerly searched by hand, by developers.

While the current implementation of Leo relies on the Dandelion compiler to optimize LINQ-based streaming workloads on NVIDIA GPUs, the same techniques and insights can be applied

to optimize general data-parallel applications on any of the various GPU back-ends. Additionally, the focus of this work was to explore the challenges arising from memory access irregularity and control flow diversity in GPU codes generated by higher-level programming tools. However, Leo can be extended to perform several other kinds of dynamic optimizations, such as improving shared memory usage and/or bank conflicts, register pressure, and thread-level parallelism. We believe such a framework can effectively ameliorate many of the effort-intensive development, tuning, and optimization problems that currently characterize GPU programming.

CHAPTER VI

LIBRA: AFFINITY-AWARE WORK-STEALING FOR INTEGRATED CPU/GPU PROCESSORS

6.1 Introduction

GPUs have become pervasive in computer systems due to their ability to provide significant improvements in both energy use and performance. However, it has been a challenge for applications to effectively leverage the compute capabilities of both the CPU and GPU processors [12, 47, 57, 63]. In fact, even for platforms on which CPUs and GPU are integrated on the same die and where CPU and GPU have access to the same physical memory, the prevailing programming model has been one in which CPUs offload specific tasks to a GPU and wait for their completion before resuming the computation.

Recent hardware advances have made possible more effective, finer-grain models of combined CPU-GPU computation. Specifically, recent integrated CPU-GPU processors, such as Intel's Broadwell and Skylake processors, and AMD's Kaveri and Carrizo systems, offer hardware CPU-GPU shared virtual memory (SVM), memory coherency, and atomic operations. Such hardware support is an effective basis for realizing GPU-capable fine-grain work-stealing schedulers operating across both sets of cores.

For multi-core CPUs, work-stealing [2, 20] efficiently achieves fine-grain, instantaneous load-balancing, by enabling idle cores to steal tasks from queues of overloaded cores. A typical work-stealing implementation assigns a work-queue to each hardware thread, and distributes the workload, divided into chunks, between those queues. At runtime, each hardware thread executes work chunks from its own queue until that becomes empty, at which time it begins stealing work from

other queues. This continues until all queues are empty and the computation is complete.

While work-stealing has been extensively optimized on multi-core systems [5, 19, 41, 45, 99], little work has been done on integrated CPU-GPU processors. Efficient work-stealing on integrated CPU-GPU processors is challenging: CPUs and GPUs typically operate at different clock frequencies and have different core configurations and memory hierarchies, making their performance differ by an order of magnitude or more. As a result, while classical work-stealing enables seamless and dynamic work distribution across compute units in the presence of load imbalance, it does not always work well with such a large gap in performance.

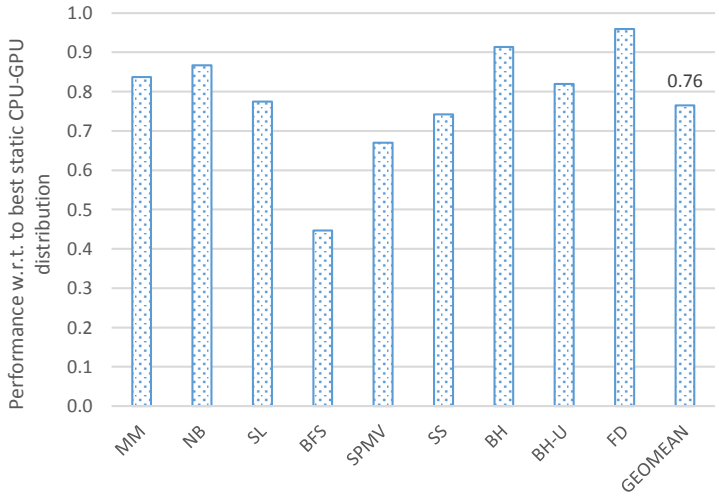

Figure 42: Performance obtained by classical work-stealing, relative to the best statically determined CPU-GPU distribution.

Figure 42 shows the performance obtained by a classical work-stealing scheduler over the best statically-determined CPU-GPU distribution. Classical work-stealing achieves only 76%, on average, and sometimes as low as 45%, of the best manually-determined CPU-GPU work distribution. Multiple factors contribute to classical work-stealing's poor performance. First, consider the situation when at the end of the execution, all queues are empty, but the GPU must remain idle while the (usually slower) CPU completes work it had already picked from its queues. The added delay

and resulting lowered performance can be significant.

Second, CPUs and GPUs have very different stealing costs. In typical work-stealing systems, the steal operation utilizes one atomic compare-and-swap (CAS) instruction, and the stealing cost scales with its latency. To show this difference in stealing costs, we had the CPU steal 64K chunks of work repeatedly, then compared its performance to the GPU doing the same. In both cases, an identical amount of work (64K) was stolen by a single work-group, and no work was actually executed. Figure 43 shows the relative performance of the CPU over that of the GPU for varying CPU frequencies. The GPU was was set to its maximum frequency (800 MHz). At their respectively maximum frequencies (CPU=2400 MHz, GPU=800 MHz), the CPU steals approximately $9\times$ faster than the GPU, and even when the CPU is set to the same frequency as the GPU, the CPU is still about $4\times$ faster.

CPUs and GPUs have multiple architectural differences that impact their stealing costs. Multicore CPUs typically offer a small number of powerful cores, while GPUs have a large number of simpler, Single Instruction Multiple Data (SIMD) cores. GPUs typically operate at lower clock frequencies. While CPUs offer excellent single-threaded compute performance, GPUs achieve high performance compute throughput by executing thousands of concurrent threads. Moreover, CPUs typically have lower-latency paths to local caches as well as lower-latency atomic instructions than do GPUs, but GPUs effectively hide such latency by continually switching threads on any long-latency event.

In a classical work-stealing scheduler, which does not take into account such a huge disparity in stealing costs, the CPU is likely to grab more work than its ideal distribution, resulting in additional overhead toward the end of the work-stealing execution. For this reason, a classical work-stealing scheduler will fail to provide good performance across diverse workloads.

We describe *Libra*, the first implementation of a fully-integrated, affinity-aware CPU-GPU work-stealing scheduler. Libra utilizes the hardware shared virtual memory (SVM), memory coherency, and CPU-GPU atomics support of Intel's Core processors. It incorporates several

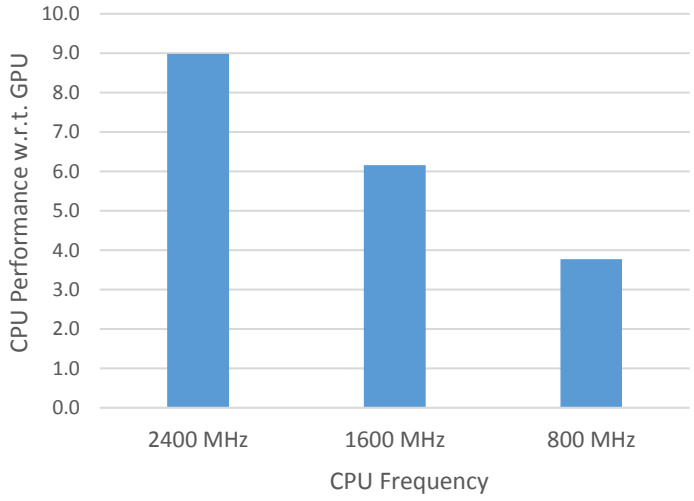

Figure 43: Normalized CPU performance of work-stealing scheduler overhead over GPU performance, as tested with a null workload, and across varying CPU frequencies. The GPU frequency is set close to its maximum frequency (800 MHz).

implementation techniques to address the challenges discussed above. First, the scheduler uses lightweight online profiling to incorporate *device affinity* for a given workload. It distributes work across the two types of devices according to their profiled runtime performance. Second, it uses *hierarchical stealing*. While prior work has used hierarchical stealing to reduce communication costs in cluster environments [39, 45], we use it to limit the amount of stealing by a device that is less effective at executing the application's work. Libra's hierarchical stealing supports an application's affinity to one device over the other.

The specific contributions of our work in this chapter are as follows:

- We present the design and implementation of Libra, an affinity-aware work-stealing scheduling runtime for integrated CPU-GPU processors that leverages hardware support for shared virtual memory (SVM) and CPU-GPU atomics available on Intel's Core processors.

- We incorporate novel runtime techniques, such as lightweight online profiling and hierarchical stealing, to address the limitations of classical work-stealing and to do affinity-aware

distribution.

- We demonstrate the effectiveness of our affinity-aware work-stealing algorithm using a mix of both regular and irregular workloads that vary in their affinity for CPU or GPU execution. We show that our implementation outperforms classical work-stealing as well as existing CPU-GPU load-balancing techniques such as the shared-queue [77] and asymmetric online-profiling [47] approaches.

6.2 Background

6.2.1 OpenCL 2.0 and SVM

OpenCL is a parallel computing framework that supports execution of programs on a heterogeneous collection of CPUs, GPUs, and other compute devices. OpenCL allows host code to launch a function (called a *kernel*) for execution by one or more devices. Typical OpenCL kernels are data-parallel and are executed by a set of SIMD threads (each called a *work-item*) on each device. A device manages a kernel's work-items in groups called *work-groups*, which execute the kernel in a lock-step fashion. The work-items of a work-group share high-speed *local* memory and work-group execution barriers.

In integrated CPU-GPU processors, the CPU and GPU typically share a single memory controller and same physical memory. This enables the GPU and CPU to directly share data without the need for expensive data transfers over, say, a PCIe bus.

In the past, OpenCL supported a relaxed memory model where memory consistency was only guaranteed at certain synchronization points. More recently, OpenCL 2.0 and some newer integrated processors such as Intel's Broadwell and AMD's Kaveri have added hardware support for shared virtual memory (SVM), and have added cache coherency protocols to ensure correct sharing between CPU local caches and the architecturally separate GPU local caches. This SVM support enables directly sharing pointer-containing data structures like trees or lists between a CPU and

GPU. Furthermore, if OpenCL's optional CPU-GPU atomic operations are supported (as in, for example, Broadwell processors), those operations can be used to provide fine-grained control of memory consistency, enabling the host and GPU to concurrently read and update the same memory locations with consistency provided by SVM atomics in addition to the consistency provided at synchronization points.

This hardware support for CPU-GPU SVM, memory coherency, and atomics enables the development of sophisticated scheduling techniques such as work-stealing schedulers that exploit all CPU and GPU cores to execute parallel programs. Our work-stealing system is built using OpenCL 2.0 features to implement efficient dynamic work distribution between the CPU and GPU.

6.2.2 Work-stealing in C++11

On the CPU, work-stealing is a well-known and highly effective technique for dynamically distributing the tasks of parallel programs. It relieves the programmer from the burden of choosing an optimal static work partition. It is a key technology, for example, of the Cilk programming framework [20]. Work-stealing schedulers assign a work-queue to each hardware thread, and the work, typically divided into chunks, is distributed between those queues. At runtime, each hardware thread or computation unit processes items from its work queue until the queue becomes empty, and then it starts stealing chunks of work from other queues until all queues are empty and the computation is complete.

Our CPU-GPU work-stealing algorithms are a simplified version of Chase and Lev's lock-free work-stealing algorithm [26] for shared-memory CPU multiprocessor environments. Their implementation allows each hardware thread, or *worker thread*, to manage its own doubly-ended queue (deque). A worker may push or pop tasks from the bottom of its own deque, but other workers can only steal tasks from the top when their own deques become empty. Assuming sequentially consistent memory operations, the Chase-Lev implementation only requires one atomic compare-and-swap (CAS) operation per steal, no CAS operation on push, and no CAS operation on pop

except when the deque has only one item left, for which other workers may contend when attempting to steal. Our implementation was simplified by supporting only fixed-size deques in order to efficiently use OpenCL fine-grain buffer SVM.

We show our OpenCL 2.0 implementation of the push, pop and steal routines in Listing 6.1. The push routine pushes items to one end, the bottom, of the deque. The pop routine tries to remove the item from the bottom of the deque. If the deque is not empty and the item to be popped is not the last one in the deque, the pop routine returns the item, updating the counters appropriately. Otherwise, it performs an atomic CAS operation. The steal routine checks whether the deque is empty, and if not, it tries to obtain the item from the other end, top of the deque, using an atomic CAS operation.

Listing 6.1: Chase-Lev deque in OpenCL 2.0

```
void push(int item, WSQueue *ws_q)
{
      int b = ws_q->bottom;
      ws_q->array[b % SIZE] = item;
      ws_q->bottom = b+1;
}

int pop(global WSQueue *ws_q)
{
      int bottom = ws_q->bottom;
      bottom = bottom - 1;
      ws_q->bottom = bottom;

      atomic_work_item_fence(CLK_GLOBAL_MEM_FENCE,
         memory_order_seq_cst, memory_scope_all_svm_devices);
```

```
        int top = ws_q->top;
        int size = bottom - top;
        if (size < 0) {
                ws_q->bottom = top;
                return EMPTY;
        }



        int item = ws_q->array[bottom % SIZE];
        if (size > 0) return item;

        if (!atomic_compare_exchange_strong((volatile atomic_int*)&(
           ws_q->top), &top, top+1))
        {
                item = EMPTY;
        }
        ws_q->bottom = top+1;
        return item;
}

int steal(global WSQueue *ws_q)
{
        int top = ws_q->top;
        atomic_work_item_fence(CLK_GLOBAL_MEM_FENCE,
           memory_order_seq_cst, memory_scope_all_svm_devices);
        int bottom = ws_q->bottom;
```

```
        int size = bottom - top;
        if(size <= 0) return EMPTY;

        int item = ws_q->array[top % SIZE];
        if (!atomic_compare_exchange_strong((volatile atomic_int*)&(
           ws_q->top), &top, top+1))
        {
                return ABORT;
        }
        return item;
}
```

6.3 Our Approach

The goal of our work-stealing scheduling runtime is to balance data-parallel computation across the cores of CPU and GPU. In order to achieve this, our runtime must be able to: (1) map high-level parallel computations to OpenCL work-groups, (2) bind the work-groups to physical cores; (3) assign work-stealing deques to work-groups; and (4) finally, perform work-stealing among the work-groups. In this section, we first describe how we implement the above steps in a classical work-stealing setting. We then present our extensions to better deal with device bias and application-level irregularity (called *affinity-aware* work-stealing).

In our runtime, each work-group maintains its own work-stealing deque. We designate a single work-item in each work-group to pop or steal work for the entire work-group, which requires the use of shared local memory and work-group barriers.

Our work-stealing runtime starts by launching a different number of work-groups on each device, based on the underlying hardware configuration. For the CPU, we launch one work-group per hardware thread, and for the GPU, we launch one work-group per EU.

In OpenCL, a work-item is identified by its unique index in the index space defined by the host program when it launches the kernel for execution on a device. We represent a *chunk* of work, the work performed by an entire work-group, by its starting index, which we call its *workId*. Initially, the chunks of work are evenly distributed across the work-stealing deques of each device.

The following two sections describe two work-stealing algorithms. The first is a Classical work-stealing algorithm, a straightforward implementation of CPU-GPU work-stealing. We then describe our affinity-aware Libra work-stealing algorithm.

6.3.1 Classical Work-stealing Algorithm

Our *classical* work-stealing scheduler extends the Chase-Lev deque approach to the OpenCL execution model. Our algorithm provides the basic skeleton for a work-stealing scheduler that supports both CPU and GPU cores, but is naive about the heterogeneity of the underlying devices and any runtime bias a workload may exhibit toward one device.

```
Input : deques: Pointer to work-stealing deques; NUMQUEUES: Total number of work-stealing deques; origKernel:
        Function pointer to original application kernel
tId ← getLocalThreadId();
groupId ← getWorkgroupId();
local workId ← EMPTY;
i ← 0;
while true do
    if tId = 0 then
        workId ← pop(deques[groupId]);
        if workId = EMPTY then
            while i < NUMQUEUES do
                j = (groupId + i + 1) mod NUMQUEUES;
                workId ← steal(deques[j]);
                if workId ≠ EMPTY then
                    break;
                end
                i++;
            end
        end
    end
    local memory barrier;
    if workId = EMPTY then
        break;
    end
    call origKernel on chunk starting at workId;
end
```

Algorithm 1: Classical CPU-GPU work-stealing algorithm

As shown in Algorithm 1, the work-item with the local thread index of zero in its work-group is designated to get work for the entire work-group. If it is unable to find work in its own deque, it tries to steal work from other deques. In doing its search, it starts from the deque following its own to minimize contention. As soon as that work-item finds a chunk of work, it exits from the loop. We use a local barrier at that point to ensure that the selected value of *workId* is visible to all work-items of the work-group before they resume execution. Once that happens, each work-item calls the original application kernel. In this way, the entire chunk of work is executed by the work-group. The work-group continues to fetch and execute work chunks until all deques are empty. Figure 44(left) provides a visual depiction of how the classical work-stealing algorithm pops and steals chunks of work.

Although classical work-stealing is able to deal with application load imbalance, it neither mitigates the contention between CPU and GPU threads as they try to steal work from the same deques, nor does it address the disparity in stealing costs between CPU and GPU cores.

6.3.2 Libra Work-stealing Algorithm

Classical work-stealing assumes that steal has a uniform cost across all workers. In a CPU-GPU heterogeneous environment, however, this assumption is no longer valid. When the CPU and GPU contend to steal the same chunk of work, it is highly likely that the CPU will succeed even when the CPU is not the fastest device. This can severely impact performance.

In addition, since the CPU and GPU in an integrated processor have different execution characteristics, many applications will have a strong affinity to one device over the other. For example, the applications *Matrix-Mul (MM), N-Body (NB)* and *Skip-List (SL)* perform best when mostly executed on the GPU, and so are GPU-biased. One the other hand, *Barnes-Hut (BH), Substring-Finder (SS)* and *Face-Detect (FD)* are CPU-biased. The runtime behavior of the same application can also vary with different inputs. Determining an application's bias to a particular device, therefore, must be determined at runtime.

Several techniques can be used to find an application's device bias: machine learning [40], instrumentation [35], offline profiling [63], and online profiling [47]. As described next, we use lightweight online profiling to determine device bias at runtime for a given workload-input pair. This lets us optimize initial work placement and stealing.

6.3.2.1 Lightweight Online Profiling

Initially, we add a single chunk of work to each device's deques. We measure the time it takes for each work-group on a particular device to complete that single work chunk, and calculate each device's aggregate throughput. We then redistribute the remaining work according to the workload's runtime affinity to the devices. How we choose the distribution is covered in more detail in Section 6.3.2.2.

After redistributing the work, we relaunch the kernel and let the application run to completion. The overhead for online profiling consists of throughput calculation, distribution of the remaining work, a second kernel launch, and the additional time due to *slack*, the difference in profiling times of the two devices. For most applications, a kernel launch is a small fraction of the total kernel execution time. The average overhead due to our lightweight online profiling across all of our workloads is a negligible *0.85%* of the total kernel execution time.

In Libra, the online profiling results merely serve as a hint to indicate affinity to a certain device. We rely on dynamic work-stealing to correct any load imbalance due to inaccuracies in our online profiling. In particular, we do not need to perform repeated profiling for irregular, input-dependent workloads, as is done in [47].

6.3.2.2 Device-Aware Distribution

After finishing online profiling, we redistribute work between the two devices according to a formula, which we derive as follows. Let the computational rate for the CPU on the initial profiling work chunk be R_c, and the rate for the GPU be R_g. Furthermore, let N denote the number of

remaining chunks of work. We want to determine the value of α, the ideal distribution ratio to the GPU, as well as $1 - \alpha$, the remaining ratio to the CPU. For best execution time, we want both devices to complete execution at the same time. That is,

$$\frac{N\alpha}{R_g} = \frac{N(1-\alpha)}{R_c} \tag{3}$$

This can be rewritten to give us α:

$$\alpha = \frac{R_g}{R_g + R_c}$$

6.3.2.3 Hierarchical Stealing

While Libra's affinity-aware placement helps with heterogeneous work-stealing, it is not sufficient. On biased workloads, worker threads on the unbiased device may still steal too much work from the other device, significantly slowing down computation. To address this, we introduce *hierarchical stealing*: worker threads on each device first steal only from deques on the same device. Only when all deques on its own device are empty is a worker thread allowed to steal from the other device's deques.

6.3.2.4 Putting It Together

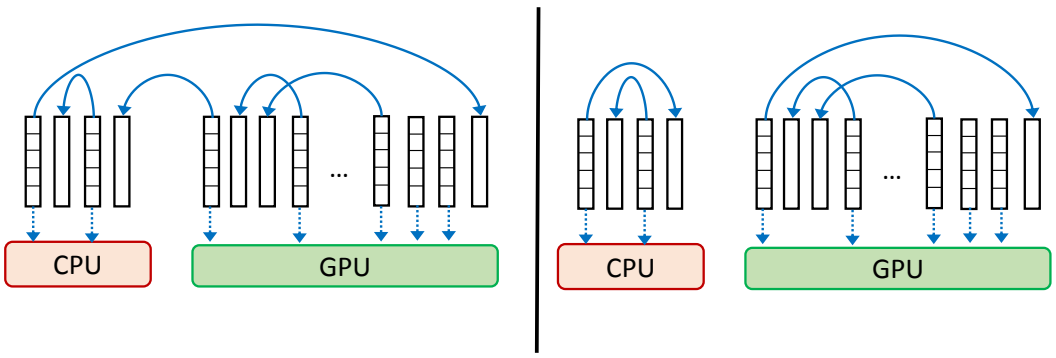

Figure 44: Classical work-stealing (left) and Libra's affinity-aware work-stealing that uses hierarchical stealing (right).

```
Input : deques: Pointer to work-stealing deques; myOffset: Offset into the deques structure where my own device's deques
        start; otherOffset: Offset into the deques structure where the other device's deques start; myQueues: Total number
        of work-stealing deques on my own device; otherQueues: Total number of work-stealing deques on the other device;
        origKernel: Function pointer to original application kernel; profiling: Flag to indicate whether profiling is enabled or
        disabled;
tid ← getLocalThreadId();
groupid ← getWorkgroupId();
local workid ← EMPTY;
while true do
    if tid = 0 then
        workid ← pop(deques[groupid]);
        if workid = EMPTY then
            for i ← myOffset to (myOffset + myQueues) do
                k = (groupId + i + 1) mod (myOffset + myQueues);
                workid ← steal(deques[k]);
                if workid ≠ EMPTY then
                    break;
                end
            end
            if workid = EMPTY then
                for i ← otherOffset to (otherOffset + otherQueues) do
                    k = (groupId + j + 1) mod (otherOffset + otherQueues);
                    workid ← steal(deques[k]);
                    if workid ≠ EMPTY then
                        break;
                    end
                end
            end
        end
    end
    local memory barrier;
    if workid = EMPTY then
        break;
    end
    call origKernel on chunk starting at workId;
    if profiling = TRUE then
        break;
    end
end
```

Algorithm 2: Adaptive CPU-GPU work-stealing algorithm

The details of our affinity-aware algorithm are shown in Algorithm 2. Prior to launching the OpenCL work-stealing kernel, the variables `myOffset`, `otherOffset`, `myQueues`, and `otherQueues` are set appropriately for each device. Additionally, the `profiling` flag is enabled only the first time a particular kernel is launched, and disabled afterwards. A single work-item in each work-group tries to obtain work for that work-group, first by popping from its own deque. If its own deque is empty, it tries stealing from deques belonging to the same device, starting from the deque following its own. If it still fails to find work, it tries to steal from deques belonging to the other device. Once it has found work, a local barrier is performed to ensure the `workId` is visible to all work-items in the work-group, and then each work-item calls the original application kernel. Figure 44(right) depicts hierarchical stealing in Libra's affinity-aware work-stealing scheduler.

6.4 Implementation

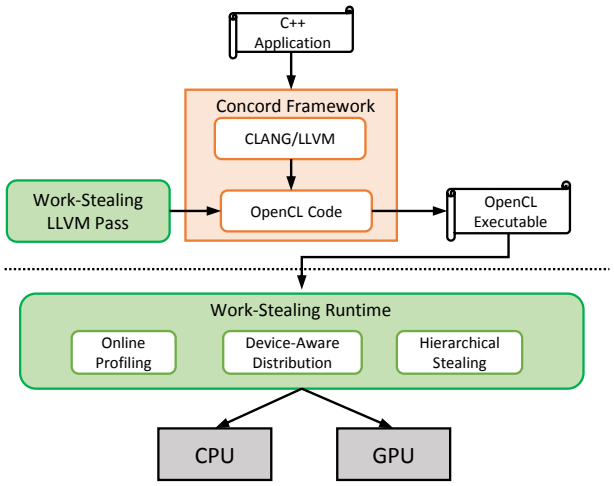

Figure 45: Concord [16] compiler framework with seamless work-stealing for C++ applications. The new components implemented for our work-stealing runtime are shown in green.

To implement our Libra work-stealing runtime, we extended the existing Concord [16] open-source heterogeneous C++ programming framework. Concord makes use of OpenCL to let general-purpose, object-oriented C++ applications take advantage of combined CPU and GPU execution. It supports two data-parallel constructs, a parallel-for loop and a parallel-reduction loop, that are modeled after ones provided by Thread Building Blocks (TBB) [2]. We modified Concord to use Libra to implement these loops. Figure 45 shows Libra's components.

Concord uses the CLANG and LLVM infrastructure to compile C++ programs into OpenCL kernels. A compiler pass identifies the Concord loop bodies and generates OpenCL kernel code for them. For Libra, we added an LLVM pass that generates work-stealing wrapper code around the OpenCL kernels. We change the original kernel to an internal function, which is invoked by our wrapper kernel when chunks of work need to be processed. Our wrapper code implements the algorithms discussed in Section 6.3, and takes additional kernel parameters such as the pointer to the work-stealing deques and a profiling flag. Listings 6.2 and 6.3 show the original application kernel and the transformed work-stealing kernel, respectively.

We use OpenCL 2.0 APIs to allocate a shared SVM buffer that is shared by the CPU and GPU. This buffer stores all the necessary shared data structures needed to implement the work-stealing deques: this is primarily the arrays holding indices representing the individual work chunks and profiling counters that track of number of pop and steal operations per deque.

Listing 6.2: Original Application OpenCL Kernel

```
__kernel void origAppKernel(void *applicationContext, uint offset)
{
  //application-specific code
}
```

Listing 6.3: Work-Stealing OpenCL Kernel

```
__kernel void workstealingKernel(void *applicationContext, uint
    offset, WSQueue *deques, uint myOffset, uint myNumQueues, uint
    oOffset, uint oNumQueues, uint profiling)
{
  __local int workidx;
  int tid = get_local_id(0);
  int groupid = get_group_id(0);

  while(1)
  {
    if(tid == 0)
    {
          //pop from own deque or steal from others
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if(workidx == EMPTY) break;
    origAppKernel(applicationContext, workidx + offset);
      if(profiling) break;

  }
}
```

6.5 *Evaluation*

We now present an experimental evaluation of our Libra work-stealing scheduler.

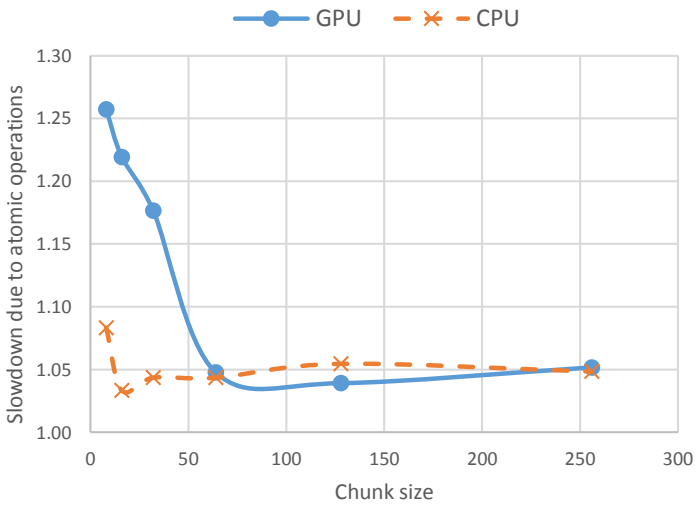

Figure 46: Slowdown on a compute-bound micro-benchmark from stealing with varying chunk sizes on the CPU and GPU.

Table 9: Benchmark characteristics for Libra

Abbrev	**Name**	**Input**	**Static Oracle (CPU/GPU) Dist.**
MM	Matrix-Multiply	matrix 2048 by 2048	0/100
NB	N-Body	4096 bodies	0/100
SL	Skip-List	10M keys	10/90
BFS	Breadth-First Search	G3 circuit sparse graph (1.6M nodes, 3M edges)	20/80
SPMV	Sparse Matrix Vector Multiply	Subset of Clueweb graph (100M nodes, 2B edges)	40/60
SS	Substring-Finder	28657 items	60/40
BH	Barnes-Hut	1M bodies	50/50
BH-U	Barnes-Hut (unsorted)	1M bodies	60/40
FD	Face-Detect	image 3000x2171	100/0

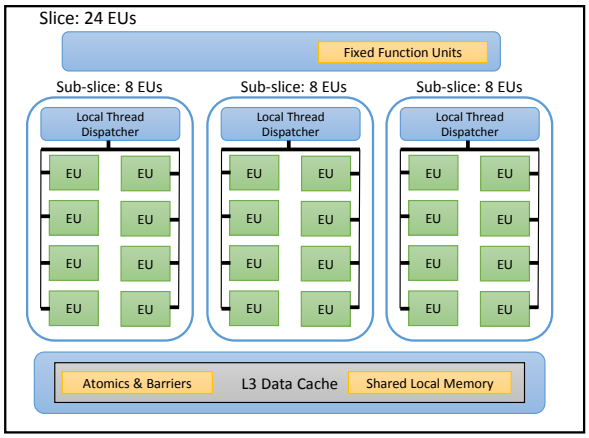

Figure 47: Intel's HD Graphics 5300 Compute Architecture [1]

6.5.1 Environment

We performed our evaluation on a laptop system with 8GB memory running 64-bit Windows 8. It includes a 1.2GHz Intel Broadwell (5th generation) Core-M 5Y71 processor with two CPU cores and hyper-threading enabled. Its integrated GPU, an Intel HD Graphics 5300, has 24 execution units (EUs), each with seven 16-wide SIMD hardware threads, with a maximum clock speed of 900 MHz.

Our GPU contains 24 execution units (EUs) that resemble CPU cores. These EUs are organized into an assembly of three modular sub-slices, each with 8 EUs (see Figure 47). Each sub-slice also contains its own local instruction caches and thread dispatcher unit. Sub-slices are combined into slices. Aside from grouping sub-slices, the slice integrates additional logic for thread dispatch routing, a banked level-3 cache, a smaller but highly-banked shared local memory structure, and fixed function logic for atomics and barriers.

Our evaluation used a diverse set of benchmarks, spanning a spectrum of application domains and varying runtime characteristics such as regular and irregular workloads. By "irregular", we mean execution imbalance across different chunks of computation. The benchmarks, their input sets, and static Oracle CPU-GPU distributions are listed in Table 9.

6.5.2 Impact of Work-Stealing Chunk Size

To understand the granularity at which our work-stealing runtime could be effective, we experimented with stealing different amounts of work. We used a compute-bound micro-benchmark with a kernel that did a minuscule amount of computation (a fixed number of floating point operations) to emphasize the overhead of the CPU and GPU atomic and other operations required for work-stealing. Figure 46 shows the slowdown from stealing different chunk sizes of work. The cost of stealing is ameliorated over larger chunks of work, but interestingly, it is much higher on the GPU than the CPU for smaller chunk sizes. On our platform, a chunk size of at least 64 elements is required to ameliorate the GPU's stealing cost, while even a chunk size as small as 16 elements yields good performance on the CPU. Based on this study, we chose 64 as our chunk size.

6.5.3 Mitigating Contention in Classical Work-Stealing

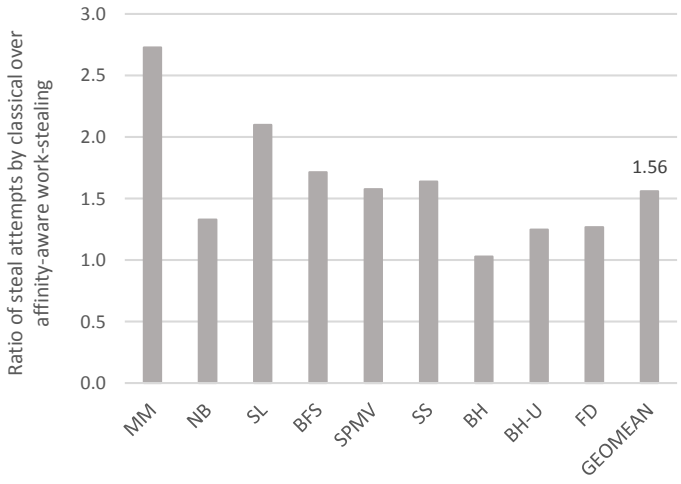

Figure 48: Ratio of steal attempts by classical work-stealing over Libra. Larger values indicate more stealing contention with classical work-stealing.

The goal of our affinity-aware heuristics is to mitigate CPU-GPU contention and deal with the disparity in the stealing costs on the CPU and GPU. Figure 48 shows the ratio of total steal attempts performed by our classical work-stealing scheduler versus those performed by our affinity-aware

Libra work-stealing scheduler. A higher ratio indicates that classical scheduling increases the total number of steal attempts, thereby increasing the total stealing overhead. By using lightweight online profiling to obtain an affinity-aware initial work distribution, and then allowing a device to steal work only from its own deques until all of those deques are empty, our affinity-aware algorithm reduces the total number of steal attempts by 56%, on average, across all workloads.

6.5.4 Performance of the Libra Work-Stealing Scheduler

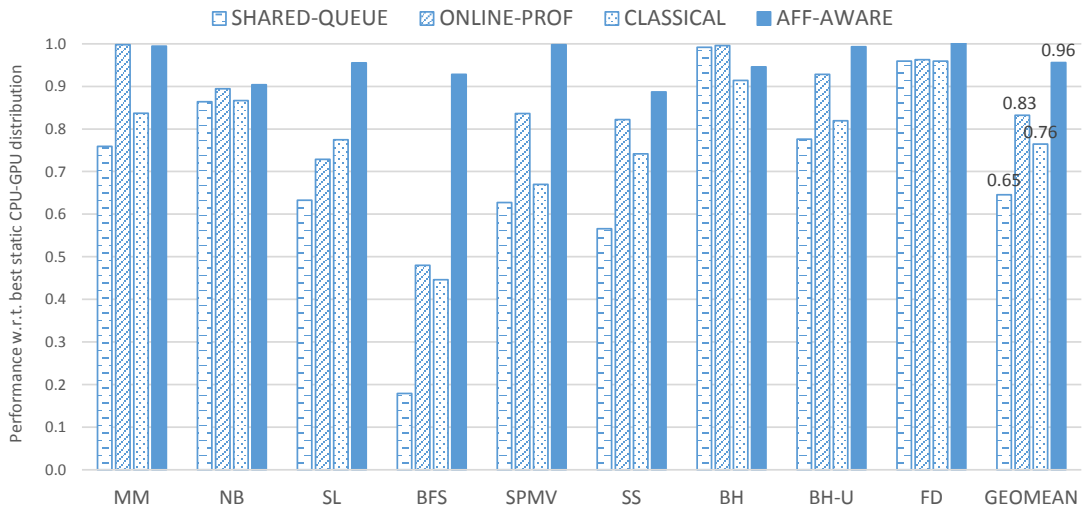

Figure 49: Performance relative to the static Oracle CPU-GPU work distribution, for shared-queue scheduling, online profiling-based scheduling, classical work-stealing and affinity-aware work-stealing.

We next compare the performance of Libra and other heterogeneous schedulers compared to a baseline performance using a near-ideal Oracle static work distribution. The Oracle static distribution is obtained via exhaustive search: we ran each benchmark with different fixed CPU-GPU work partitions, varying the percentage of work given to the CPU from 0% to 100% in steps of 10%, and selected the best-performing CPU-GPU work distribution.

The schedulers we compare include our Libra and the classical schedulers as well as the following:

Shared-Queue: The shared-queue FluidiCL OpenCL runtime [77] implements heterogeneous load-balancing using CPU-GPU cooperation to cope with irregular workloads and devices with different execution characteristics. FluidiCL supports GPUs without SVM including most discrete GPUs. In contrast, our shared-queue implementation uses SVM CPU-GPU atomics, while FluidiCL requires explicit data transfers and management to maintain CPU-GPU coherency in systems where hardware SVM support is not available.

Asymmetric Online Profiling: We implemented an asymmetric online profile-based scheduler similar to that described in [47]. It uses the profiling technique that achieved the best performance in [47], *ASYM+SIZE*, that profiles using half the work-items. However, our asymmetric online scheduler uses hardware SVM support instead of a proxy thread to offload work to the GPU. Note that the asymmetric online profile-based scheduler differs from Libra by doing online profiling then statically dividing the remaining work based on those results. Libra, on the other hand, does online profiling to determine device affinity and to place work at the start of work-stealing, then lets work-stealing dynamically determine the best subsequent distribution.

Figure 49 shows the performance of each scheduler relative to the near-ideal static CPU-GPU distribution. Classical work-stealing achieve only 76%, on average, of the performance of the Oracle static distribution. This is not surprising since classical does not address the challenges associated with heterogeneous execution and dynamic runtime characteristics.

Our affinity-aware work-stealing scheduler overcomes these challenges. Overall, Libra achieves 96% performance of the Oracle static distribution. Libra also outperforms both the shared-queue and asymmetric online profiling schedulers. While asymmetric online profiling usually works well [47], it inherently suffers from any inaccuracies in profiling information. This is especially a problem for irregular workloads in which the initial iteration space is not representative of later computation stages. Even with profiling half of the iteration space, for highly irregular data sets such as those used for *BFS*, *SPMV*, and *SL*, asymmetric online profiling does not perform well, while affinity-aware work-stealing can dynamically adapt even to such highly irregular workloads.

Figure 50 shows the CPU-GPU work distribution achieved by our affinity-aware and classical work-stealing schedulers, as well as the static Oracle CPU-GPU distribution. For all workloads but *FD*, our affinity-aware Libra scheduler comes closer to the static Oracle distribution than the classical scheduler. In general, the distribution achieved by the classical scheduler is more biased toward the CPU, primarily due to the CPU's advantage in stealing.

While the static Oracle distribution for *FD* was determined to be a 0-100 CPU-GPU distribution, both Libra and the classical scheduler achieve around a 50-50 CPU-GPU distribution. *FD* is an irregular workload with unequal chunks of work. From looking at the entire CPU-GPU work distribution spectrum, the second best distribution was at 50-50 CPU-GPU distribution, and the performance difference between the two distributions was negligible. Libra and the classical scheduler may achieve a significantly different distribution than the static Oracle one, especially for irregular workloads, and still perform well.

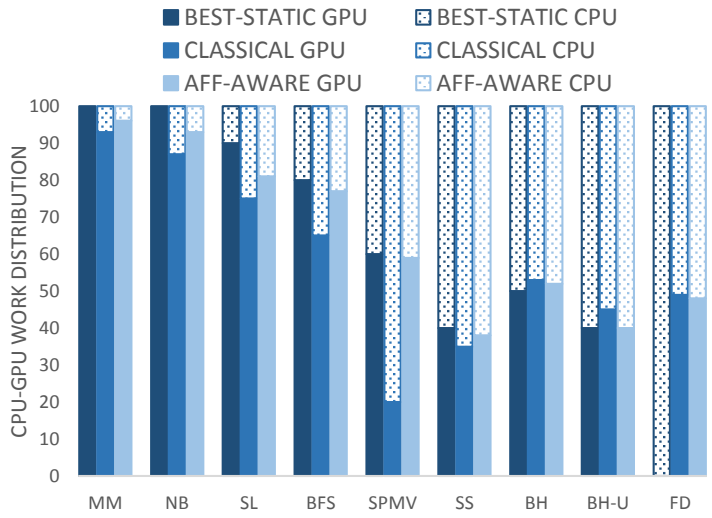

Figure 50: CPU-GPU work distribution achieved by the static Oracle, classical work-stealing, and affinity-based Libra work-stealing schedulers.

Table 10: BFS input graphs selected for Libra

Name	Input	Static Oracle (CPU/GPU) Dist.
BFS-EC	1M nodes, 2M edges	10/90
BFS-G3	1.6M nodes, 3M edges	20/80
BFS-LD	1M nodes, 23M edges	30/70
BFS-KKT	2M nodes, 6.5M edges	0/100
BFS-TH	1.2M nodes, 3.7M edges	0/100

6.5.5 Dealing with Input-Dependent Behavior

Our final results show the performance of the classical and affinity-aware schedulers for a *single* application across a *varied* set of distinct inputs (Figure 51). For this study, we chose Breadth-First Search operating on a mix of highly-irregular sparse matrix graphs (Table 10). Classical work-stealing does poorly on these input graphs, achieving only about 50% of the performance of the static Oracle CPU-GPU work distribution. Our affinity-aware work-stealing scheduler, however, does consistently well, even across distinct inputs, achieving on average 92% performance of the static Oracle distribution.

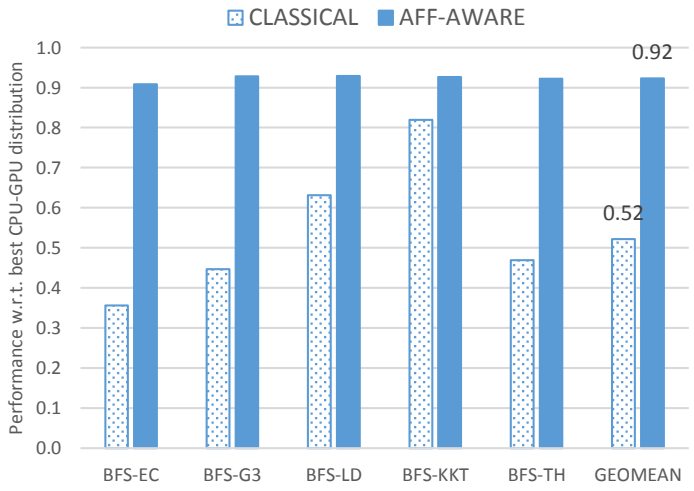

Figure 51: Performance with respect to static Oracle CPU-GPU work distribution, for the Breadth-First Search (BFS) application across distinct inputs.

In summary, our affinity-aware Libra work-stealing scheduler overcomes limitations of classical work-stealing for integrated CPU-GPU systems, performs on par with the near-ideal Oracle static distribution, and outperforms the other heterogeneous load-balancing approaches, shared-queue and asymmetric online profiling.

6.6 Related Work

Work-stealing and Optimizations. Randomized work-stealing for shared-memory CPU multiprocessors was originally introduced in the Cilk programming framework [20]. It is based on double-ended queues (deques) designed to minimize contention by doing synchronization only when stealing is needed. A state-of-the-art CPU implementation of the work-stealing deque was described by Chase and Lev [26] and is used in many runtimes and libraries including Intel's TBB. A correctness proof for an optimized implementation of the Chase-Lev concurrent deque appeared in [56].

Improving various aspects of work-stealing scheduling has been an important research area. SLAW [41] implements an adaptive scheduler that incorporates locality-aware policies. Hot-SLAW [45] presents a topology-aware hierarchical work stealing strategy that exploits locality in distributed memory systems. The underlying idea is for each worker to choose its victim (hierarchical victim selection), or the amount of work to steal (hierarchical chunk selection), based on its locality. In our system, we exploit hierarchical stealing to support device affinity. A-Steal [5] is an adaptive scheduler that considers parallelism feedback at scheduling time. AdaptiveTC [99] utilizes adaptive thread management to improve performance, while BWS [19] addresses throughput and fairness issues in time-sharing multi-core systems. Kumar et. al. [54] applied run-time techniques such as dynamic compilation and speculative optimization to further reduce overheads in X10. Hermes [85] proposed an energy-efficient work-stealing runtime that executes different threads at different speeds to maximize energy savings with minimum performance loss.

Work-stealing or load-balancing with GPUs. Several recent papers describe work-stealing

with discrete GPUs [27, 39, 89]. Their key idea is to extend standard work-stealing with restrictions on stealing: since the GPU cannot initiate communication with the CPU or steal itself, only approaches where the CPU pushes work to GPUs can address load imbalance. Such approaches reduce CPU performance since the CPU has to act on behalf of the GPU workers.

A shared-queue approach, where the GPU and CPU operate on different ends of a single queue, is proposed in [77]. The FluidiCL OpenCL runtime implements CPU-GPU cooperation to dynamically distribute work in systems without SVM. As a result, it lacks the fine-grained load-balancing possible with work-stealing in systems having shared memory.

Cederman et al. [25] address load-balancing across different cores on a discrete GPU, while [29] provide load-balancing by running persistent kernels that communicate via task-queues with the host. [52] abstracts GPUs as a single virtual device to support load-balancing across multiple GPUs, and a performance prediction model of multiple GPUs is presented in [90].

With the hardware CPU-GPU SVM, coherency, and atomic support on Intel Broadwell and AMD's Kaveri systems, our system is the first implementation, to the best of our knowledge, of a real work-stealing scheduler across CPU and GPU cores.

Heterogeneous CPU-GPU Execution. Heterogeneous execution of applications using both CPUs and GPUs has been studied extensively [12, 40, 47, 57, 63]. Qilin [63] is an API and runtime to support heterogeneous execution of applications using offline profiling. StarPU [12] supports user-defined policies for dynamic workload scheduling. SKMD [57] enables execution of a single kernel on multiple devices using static kernel analysis to determine workload distribution. Kaleem et al. [47] use online profiling to determine the best CPU-GPU workload distribution for an application's execution. Dandelion [88] implements a uniform, high-level sequential programming model across a diverse array of execution units, including CPUs and GPUs. Ravi et al. [84] use work-sharing to distribute work between the CPU and a discrete GPU. Scogland et al. [92] present several scheduling techniques for systems with discrete devices.

Unlike previous work, our affinity-aware Libra work-stealing scheduler effectively implements

fine-grained CPU-GPU work distribution using a combination of hardware CPU-GPU SVM, lightweight online profiling, and hierarchical stealing to accommodate both device architectural differences and application execution characteristics.

6.7 Chapter Summary

With hardware support for CPU-GPU shared virtual memory and atomics, it is now possible to perform true work-stealing on integrated CPU-GPU processors. We describe the implementation of Libra, an affinity-aware work-stealing scheduler for such integrated processors that efficiently distributes work across all CPU and GPU cores for improved load balance. Our study demonstrates that classicalwork-stealing does not effectively address either a significant performance imbalance or a much different stealing cost by the CPU and GPU. To deal with these challenges, we augmented our work-stealing algorithm with lightweight online profiling to enable device-aware distribution and hierarchical stealing. Our results demonstrate that our affinity-aware Libra work-stealing scheduler outperforms classical work-stealing by up to 2$\times$ and on average by 20%. It also improves over existing load-balancing techniques, such as shared queue and online profiling.

CHAPTER VII

CONCLUSION

This chapter presents final conclusions drawn for runtime specialization for heterogeneous CPU-GPU resources.

7.1 Summary

This dissertation addresses challenges associated with input-dependent, time-varying runtime behaviors of the diverse set of applications targeted for heterogeneous computing, as well as the architectural differences of CPU and GPU resources to achieve greater efficiency via runtime specialization. We propose and present a dynamic instrumentation engine for GPU-based platforms, *Lynx*, to gain real-time insights into application behavior. We integrate Lynx into runtime frameworks, such as *Luminar* and *Leo*, which provide novel dynamic resource management and optimization methods, respectively, to improve performance and throughput of both regular and irregular applications. Additionally, we leverage the shared virtual memory support present in today's integrated CPU-GPU processors to propose a specialized, affinity-aware work-stealing scheduler, *Libra*, which improves over classical work-stealing by incorporating runtime application characterization and architectural CPU-GPU differences. Our evaluation incorporate a wide array of applications, focusing on highly input-dependent, irregular workloads such as graph applications, but also including embarrassingly parallel, compute-intensive, GPU-bound applications as well as CPU-biased workloads.

7.2 Contributions

The contributions of this dissertation may be summarized as follows:

- Dynamic instrumentation support for GPU-based platforms
- Instrumentation-driven resource management and scheduling to improve throughput and performance of applications running on integrated CPU-GPU platforms
- Instrumentation-driven data layout optimizations to improve performance of applications running on GPUs
- Affinity-aware work-stealing to provide effective load-balance across CPU and GPU cores for integrated CPU-GPU processors

7.3 Future Work

While we have demonstrated the quantitative benefits of runtime specialization in improving performance for a diverse set of data-intensive applications running on heterogeneous CPU-GPU platforms, there are several interesting extensions to our work, which have not yet been thoroughly explored, that highlight the potential for runtime specialization in today's computing environments.

Novel Instrumentation-Driven Systems Abstractions

In this work, we have explored dynamic resource management and optimizations methods, driven by online GPU instrumentation. In general, the Lynx dynamic instrumentation framework provides significant opportunities for incorporating a variety of GPU systems abstractions seamlessly and transparently. For example, while GPUs have become primary processing engines in server and cloud computing environments, cloud providers are still not able to provide fine-grained GPU sharing to end-users, even though such sharing is possible for CPUs. Previous work has attempted to address shared GPU compute usage in different ways [42, 66], but have not been as effective in enforcing fine-grained service-level objectives (SLOs) due to limited control over closed-vendor, hardware thread scheduling in today's GPUs. GPU dynamic instrumentation can enable software mechanisms for fine-grained GPU time-sharing, by incorporating yielding mechanisms at smaller execution granularities (such as thread-block level, versus kernel-level).

Similarly, software-based check-pointing mechanisms for accelerator-based platforms have been proposed [48]. Such mechanisms can leverage instrumentation to provide transparency (i.e. not requiring source code modifications and therefore can work with application binaries) and portability (i.e. can target multiple GPU back-ends). Finally, in applications where precision and correctness is a necessity, dynamic instrumentation can be utilized to provide software abstractions for reliable operation [59].

Energy-Aware Heuristics

Energy efficiency is now a top design goal for all computing systems, from fitness trackers and tablets, where it affects battery life, to cloud computing centers, where it directly impacts operational cost, maintainability, and environmental impact. Much of our work has focused on improving platform throughput and application performance. While we have noted improvements in energy efficiency from our methods in some cases, a natural extension to our work is to explore energy-aware heuristics.

As an initial step toward this direction, we've proposed a black-box approach to energy-aware scheduling on integrated CPU-GPU systems [15]. Today's widespread integrated CPU-GPU processors combine a CPU and a GPU compute device with different power-performance characteristics. For these integrated processors, hardware vendors implement automatic power management policies that are typically not exposed to the end-user. Furthermore, these policies often vary between different processor generations. As a result, it is challenging to design a generally-applicable energy-aware runtime to schedule work onto both the CPU and GPU of such integrated CPU-GPU processors to optimize energy consumption.

We have implemented a scheduling runtime that automatically distributes work between the CPU and GPU to optimize a given energy metric. Our methodology is as follows: We first perform a one-time characterization of the processor's power use when executing different kinds of workloads (e.g., compute-intensive or memory-intensive) and measuring the power each uses. Afterwards, at the start of application execution, our scheduling runtime profiles its execution to

understand its runtime behavior. Our scheduler combines the application's runtime behavior with the processor's power characterization to distribute the application's remaining work between the CPU and GPU cores based on the energy metric being optimized. Our scheduler is user-level and automatic, and neither requires offline profiling nor detailed knowledge about the processor or its power management. It is suitable for the majority of processors produced today: ones that include a Power Control Unit (PCU) and don't allow the programmer to set component frequencies.

Distributed, Micro-Server Environments

Our present evaluation has been focused on a single CPU-GPU platform. However, the same ideas and techniques can be extended to distributed, micro-server environments, which incorporate multiple CPU-GPU processors. Application profiling will naturally need to be extended to consider network latencies and memory bandwidth requirements, but incorporating real-time application introspection will likely improve overall efficiency, even in distributed computing environments.

REFERENCES

[1] "Compute architecture of intel processor graphics."

[2] "Intel thread building blocks."

[3] "HP forges converged systems from Moonshot and Proliants." `http://www.enterprisetech.com/2013/12/09/hp-stacks-convergedsystems-moonshots-proliants/`, 2013.

[4] "Hp moonshot system: The worlds first software defined servers," 2013.

[5] AGRAWAL, K., HE, Y., and LEISERSON, C. E., "Adaptive work stealing with parallelism feedback," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, (New York, NY, USA), pp. 112–120, ACM, 2007.

[6] AMD, *AMD APP SDK*. AMD, 2.9 ed.

[7] AMD, *CodeXL*. AMD, 3.1 ed.

[8] AMD, *AMD Intermediate Language (IL)*. AMD, 2.4 ed., October 2011.

[9] ANDERSON, J. A., LORENZ, C. D., and TRAVESSET, A., "General purpose molecular dynamics simulations fully implemented on graphics processing units," *J. Comput. Phys.*, vol. 227, pp. 5342–5359, May 2008.

[10] ARIEL, A., FUNG, W. W. L., TURNER, A. E., and AAMODT, T. M., "Visualizing complex dynamics in many-core accelerator architectures," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (White Plains, NY, USA), pp. 164–174, March 2010.

[11] ARNOLD, M., FINK, S. J., GROVE, D., HIND, M., and SWEENEY, P. F., "A survey of adaptive optimization in virtual machines," in *Proceedings of the IEEE, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation*, 2005.

[12] AUGONNET, C., THIBAULT, S., NAMYST, R., and WACRENIER, P.-A., "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[13] BAGHSORKHI, S. S., DELAHAYE, M., PATEL, S. J., GROPP, W. D., and HWU, W.-M. W., "An adaptive performance modeling tool for gpu architectures," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 105–114, ACM, 2010.

[14] BAKHODA, A., YUAN, G., FUNG, W. W. L., WONG, H., and AAMODT, T. M., "Analyzing cuda workloads using a detailed gpu simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (Boston, MA, USA), pp. 163–174, April 2009.

[15] BARIK, R., FAROOQUI, N., LEWIS, B. T., HU, C., and SHPEISMAN, T., "A black-box approach to energy-aware scheduling on integrated cpu-gpu systems," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '16, ACM, 2016.

[16] BARIK, R., KALEEM, R., MAJETI, D., LEWIS, B. T., SHPEISMAN, T., HU, C., NI, Y., and ADL-TABATABAI, A.-R., "Efficient mapping of irregular c++ applications to integrated gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 33:33–33:43, ACM, 2014.

[17] BAUER, M., TREICHLER, S., SLAUGHTER, E., and AIKEN, A., "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, (Los Alamitos, CA, USA), pp. 66:1–66:11, IEEE Computer Society Press, 2012.

[18] BECCHI, M., SAJJAPONGSE, K., GRAVES, I., PROCTER, A., RAVI, V., and CHAKRADHAR, S., "A virtual memory based runtime to support multi-tenancy in clusters with gpus," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 97–108, ACM, 2012.

[19] BENDER, M. A. and RABIN, M. O., "Scheduling cilk multithreaded parallel programs on processors of different speeds," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '00, (New York, NY, USA), pp. 13–21, ACM, 2000.

[20] BLUMOFE, R. D. and LEISERSON, C. E., "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, Sept. 1999.

[21] BOYER, M., SKADRON, K., CHE, S., and JAYASENA, N., "Load balancing in a changing world: Dealing with heterogeneity and performance variability," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, (New York, NY, USA), pp. 21:1–21:10, ACM, 2013.

[22] BROWN, K., SUJEETH, A., LEE, H., ROMPF, T., CHAFI, H., and OLUKOTUN, K., "A heterogeneous parallel framework for domain-specific languages," in *20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011.

[23] BURTSCHER, M., NASRE, R., and PINGALI, K., "A quantitative study of irregular programs on gpus," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pp. 141–151, 2012.

[24] CATANZARO, B., GARLAND, M., and KEUTZER, K., "Copperhead: Compiling an embedded data parallel language," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), pp. 47–56, ACM, 2011.

[25] CEDERMAN, D. and TSIGAS, P., "On dynamic load balancing on graphics processors," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, (Aire-la-Ville, Switzerland), pp. 57–64, 2008.

[26] CHASE, D. and LEV, Y., "Dynamic circular work-stealing deque," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, (New York, NY, USA), pp. 21–28, ACM, 2005.

[27] CHATTERJEE, S., GROSSMAN, M., SBÎRLEA, A. S., and SARKAR, V., "Dynamic task parallelism with a GPU work-stealing runtime system," in *Languages and Compilers for Parallel Computing, 24th International Workshop, LCPC 2011, Fort Collins, CO, USA, September 8-10, 2011. Revised Selected Papers*, pp. 203–217, 2011.

[28] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J., LEE, S.-H., and SKADRON, K., "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Oct. 2009.

[29] CHEN, L., VILLA, O., KRISHNAMOORTHY, S., and GAO, G., "Dynamic load balancing on single- and multi-GPU systems," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, 2010.

[30] COLLANGE, S., DEFOUR, D., and PARELLO, D., "Barra, a modular functional gpu simulator for gpgpu," Tech. Rep. hal-00359342, 2009.

[31] DEVITO, Z., JOUBERT, N., PALACIOS, F., OAKLEY, S., MEDINA, M., BARRIENTOS, M., ELSEN, E., HAM, F., AIKEN, A., DURAISAMY, K., DARVE, E., ALONSO, J., and HANRAHAN, P., "Liszt: A domain specific language for building portable mesh-based pde solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 9:1–9:12, ACM, 2011.

[32] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., and WOOD, D. A., "Implementation techniques for main memory database systems," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, (New York, NY, USA), pp. 1–8, ACM, 1984.

[33] DIAMOS, G., KERR, A., YALAMANCHILI, S., and CLARK, N., "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 353–364, ACM, 2010.

[34] DOMINGUEZ, R., SCHAA, D., and KAELI, D., "Caracal: Dynamic translation of runtime environments for gpus," in *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units*, (Newport Beach, CA, USA), ACM, March 2011.

[35] FAROOQUI, N., KERR, A., EISENHAUER, G., SCHWAN, K., and YALAMANCHILI, S., "Lynx: A dynamic instrumentation system for data-parallel applications on gpgpu architectures," in *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pp. 58 –67, April 2012.

[36] FAROOQUI, N., KERR, A., DIAMOS, G., YALAMANCHILI, S., and SCHWAN, K., "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the 4th Workshop on General-Purpose Computation on Graphics Processing Units*, (Newport Beach, CA, USA), ACM, March 2011.

[37] FAROOQUI, N., ROSSBACH, C., YU, Y., and SCHWAN, K., "Leo: A profile-driven, dynamic optimization framework for gpu applications," in *Proceedings of the second USENIX conference on Timely Results in Operating Systems*, TRIOS '14, 2014.

[38] FAROOQUI, N., SCHWAN, K., and YALAMANCHILI, S., "Efficient instrumentation of gpgpu applications using information flow analysis and symbolic execution," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, (New York, NY, USA), pp. 19:19–19:27, ACM, 2014.

[39] GAUTIER, T., FERREIRA LIMA, J. V., MAILLARD, N., and RAFFIN, B., "Locality-Aware Work Stealing on Multi-CPU and Multi-GPU Architectures," in *6th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)*, (Berlin, Germany), Jan. 2013.

[40] GREWE, D., WANG, Z., and O'BOYLE, M. F. P., "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pp. 22:1–22:10, 2013.

[41] GUO, Y., ZHAO, J., CAVE, V., and SARKAR, V., "Slaw: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 341–342, ACM, 2010.

[42] GUPTA, V., SCHWAN, K., TOLIA, N., TALWAR, V., and RANGANATHAN, P., "Pegasus: Coordinated scheduling for virtualized accelerator-based systems," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2011.

[43] HONG, S., KIM, S. K., OGUNTEBI, T., and OLUKOTUN, K., "Accelerating cuda graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles*

and Practice of Parallel Programming, PPoPP '11, (New York, NY, USA), pp. 267–276, ACM, 2011.

[44] IMPACT, "The parboil benchmark suite," 2007.

[45] JAI MIN, S., IANCU, C., and YELICK, K., "Hierarchical work stealing on manycore clusters," in *In Fifth Conference on Partitioned Global Address Space Programming Models*, 2011.

[46] JIMÉNEZ, V. J., VILANOVA, L., GELADO, I., GIL, M., FURSIN, G., and NAVARRO, N., "Predictive runtime code scheduling for heterogeneous architectures," in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, (Berlin, Heidelberg), pp. 19–33, Springer-Verlag, 2009.

[47] KALEEM, R., BARIK, R., SHPEISMAN, T., LEWIS, B. T., HU, C., and PINGALI, K., "Adaptive heterogeneous scheduling for integrated gpus," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 151–162, ACM, 2014.

[48] KANNAN, S., FAROOQUI, N., GAVRILOVSKA, A., and SCHWAN, K., "Heterocheckpoint: Efficient checkpointing for accelerator-based systems," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pp. 738–743, 2014.

[49] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., and ISHIKAWA, Y., "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2011.

[50] KATO, S., MCTHROW, M., MALTZAHN, C., and BRANDT, S., "Gdev: First-class gpu resource management in the operating system," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 37–37, USENIX Association, 2012.

[51] KERR, A., DIAMOS, G., and YALAMANCHILI, S., "A characterization and analysis of ptx kernels," *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 2009.

[52] KIM, J., KIM, H., LEE, J. H., and LEE, J., "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (NY, USA), pp. 277–288, ACM, 2011.

[53] KIM, S., ROY, I., and TALWAR, V., "Evaluating integrated graphics processors for data center workloads," in *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, (New York, NY, USA), pp. 8:1–8:5, ACM, 2013.

[54] KUMAR, V., FRAMPTON, D., BLACKBURN, S. M., GROVE, D., and TARDIEU, O., "Work-stealing without the baggage," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, (New York, NY, USA), pp. 297–314, ACM, 2012.

[55] LATTNER, C. and ADVE, V., "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, (Palo Alto, California), Mar 2004.

[56] LÊ, N. M., POP, A., COHEN, A., and ZAPPA NARDELLI, F., "Correct and efficient work-stealing for weak memory models," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 69–80, ACM, 2013.

[57] LEE, J., SAMADI, M., PARK, Y., and MAHLKE, S., "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, PACT, 2013.

[58] LEE, K. and LIU, L., "Efficient data partitioning model for heterogeneous graphs in the cloud," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 46:1–46:12, ACM, 2013.

[59] LI, S., FAROOQUI, N., , and YALAMANCHILI, S., "Software reliability enhancements for gpu applications," in *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2013)*, Jan 2013.

[60] LINDERMAN, M. D., COLLINS, J. D., WANG, H., and MENG, T. H., "Merge: A programming model for heterogeneous multi-core systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (New York, NY, USA), pp. 287–296, ACM, 2008.

[61] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., and HELLERSTEIN, J. M., "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[62] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., and HAZELWOOD, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[63] LUK, C.-K., HONG, S., and KIM, H., "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (NY, USA), pp. 45–55, ACM, 2009.

[64] LUO, L., WONG, M., and HWU, W.-M., "An effective gpu implementation of breadth-first search," in *Proceedings of the 47th design automation conference*, pp. 52–55, ACM, 2010.

[65] MARS, J., TANG, L., HUNDT, R., SKADRON, K., and SOFFA, M. L., "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 248–259, ACM, 2011.

[66] MENYCHTAS, K., SHEN, K., and SCOTT, M. L., "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 301–316, ACM, 2014.

[67] MERRILL, D., GARLAND, M., and GRIMSHAW, A., "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 117–128, ACM, 2012.

[68] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R., KARAVANIC, K. L., KUNCHITHAPADAM, K., and NEWHALL, T., "The paradyn parallel performance measurement tool," *Computer, IEEE*, vol. 28, no. 11, pp. 37–46, 1995.

[69] MOSEGAARD, J. and SØRENSEN, T., "Real-time deformation of detailed geometry based on mappings to a less detailed physical simulation on the gpu," in *Proceedings of Eurographics Workshop on Virtual Environments*, vol. 11, pp. 105–111, 2005.

[70] NEWSOME, J., "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[71] NGUYEN, D., LENHARTH, A., and PINGALI, K., "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 456–471, ACM, 2013.

[72] NILAKANT, K. and YONEKI, E., "On the efficacy of apus for heterogeneous graph computation," in *Fourth Workshop on Systems for Future Multicore Architectures*, 2014.

[73] NVIDIA, *NVIDIA Compute PTX: Parallel Thread Execution*. NVIDIA Corporation, Santa Clara, California, 1.3 ed., October 2008.

[74] NVIDIA, *NVIDIA CUDA Compute Unified Device Architecture*. NVIDIA Corporation, Santa Clara, California, 2.1 ed., October 2008.

[75] NVIDIA, *NVIDIA Compute Visual Profiler*. NVIDIA Corporation, Santa Clara, California, 4.0 ed., May 2011.

[76] NVIDIA, *NVIDIA CUDA Tools SDK CUPTI*. NVIDIA Corporation, Santa Clara, California, 1.0 ed., February 2011.

[77] PANDIT, P. and GOVINDARAJAN, R., "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 273:273–273:283, ACM, 2014.

[78] PHOTHILIMTHANA, P. M., ANSEL, J., RAGAN-KELLEY, J., and AMARASINGHE, S., "Portable performance on heterogeneous architectures," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, (New York, NY, USA), pp. 431–444, ACM, 2013.

[79] PHULL, R., LI, C.-H., RAO, K., CADAMBI, H., and CHAKRADHAR, S., "Interference-driven resource management for gpu-based heterogeneous clusters," in *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, (New York, NY, USA), pp. 109–120, ACM, 2012.

[80] PODLOZHNYUK, V., "Black-scholes option pricing," *Part of CUDA SDK documentation*, 2007.

[81] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., and AMARASINGHE, S., "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Seattle, WA), June 2013.

[82] RAVI, V. T., BECCHI, M., AGRAWAL, G., and CHAKRADHAR, S., "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," in *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, (New York, NY, USA), pp. 217–228, ACM, 2011.

[83] RAVI, V. T., BECCHI, M., JIANG, W., AGRAWAL, G., and CHAKRADHAR, S., "Scheduling concurrent applications on a cluster of cpu-gpu nodes," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, (Washington, DC, USA), pp. 140–147, IEEE Computer Society, 2012.

[84] RAVI, V. T., MA, W., CHIU, D., and AGRAWAL, G., "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, (NY, USA), pp. 137–146, ACM, 2010.

[85] RIBIC, H. and LIU, Y. D., "Energy-efficient work-stealing language runtimes," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 513–528, ACM, 2014.

[86] ROMPF, T., SUJEETH, A. K., AMIN, N., BROWN, K. J., JOVANOVIC, V., LEE, H., JONNALAGEDDA, M., OLUKOTUN, K., and ODERSKY, M., "Optimizing data structures in

high-level programs: New directions for extensible compilers based on staging," in *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, (New York, NY, USA), pp. 497–510, ACM, 2013.

[87] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., and WITCHEL, E., "Ptask: Operating system abstractions to manage gpus as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 233–248, ACM, 2011.

[88] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., and FETTERLY, D., "Dandelion: a compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (NY, USA), pp. 49–68, ACM, 2013.

[89] SBÎRLEA, A., ZOU, Y., BUDIMLÍC, Z., CONG, J., and SARKAR, V., "Mapping a data-flow programming model onto heterogeneous platforms," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, (New York, NY, USA), pp. 61–70, ACM, 2012.

[90] SCHAA, D. and KAELI, D., "Exploring the multiple-GPU design space," in *IEEE International Symposium on Parallel Distributed Processing. IPDPS.*, pp. 1–12, 2009.

[91] SCHWARTZ, E. J., AVGERINOS, T., and BRUMLEY, D., "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask," in *In Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[92] SCOGLAND, T., ROUNTREE, B., CHUN FENG, W., and DE SUPINSKI, B., "Heterogeneous task scheduling for accelerated OpenMP," in *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pp. 144–155, May 2012.

[93] SEN, K., MARINOV, D., and AGHA, G., "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, (New York, NY, USA), pp. 263–272, ACM, 2005.

[94] SHENDE, S. and MALONY, A., "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, pp. 287–311, 2006.

[95] SILBERSTEIN, M., FORD, B., KEIDAR, I., and WITCHEL, E., "Gpufs: integrating file systems with gpus," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, ACM, 2013.

[96] STONE, J. E., GOHARA, D., and SHI, G., "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, pp. 66–73, May 2010.

[97] SUJEETH, A. K., LEE, H., BROWN, K. J., ROMPF, T., CHAFI, H., WU, M., ATREYA, A. R., ODERSKY, M., and OLUKOTUN, K., "Optiml: An implicitly parallel domain-specific language for machine learning.," in *ICML* (GETOOR, L. and SCHEFFER, T., eds.), pp. 609–616, Omnipress, 2011.

[98] SUNG, I., LIU, G., and HWU, W., "Dl: A data layout transformation system for heterogeneous computing," in *Innovative Parallel Computing (InPar), 2012*, p. 111, IEEE, 2012.

[99] WANG, L., CUI, H., DUAN, Y., LU, F., FENG, X., and YEW, P.-C., "An adaptive task creation strategy for work-stealing scheduling," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, (New York, NY, USA), pp. 266–277, ACM, 2010.

[100] WU, B., ZHAO, Z., ZHANG, E. Z., JIANG, Y., and SHEN, X., "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, (New York, NY, USA), pp. 57–68, ACM, 2013.

[101] WU, H., DIAMOS, G., CADAMBI, S., and YALAMANCHILI, S., "Kernel weaver: Automatically fusing database primitives for efficient gpu computation," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 12, 2012.

[102] WU, H., DIAMOS, G., SHEARD, T., AREF, M., BAXTER, S., GARLAND, M., and YALAMANCHILI, S., "Red fox: An execution environment for relational query processing on gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), pp. 44:44–44:54, ACM, 2014.

[103] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, U., GUNDA, P. K., CURREY, J., MCSHERRY, F., and ACHAN, K., "Some sample programs written in DryadLINQ," Tech. Rep. MSR-TR-2008-74, Microsoft Research, May 2008.

[104] ZHANG, E., JIANG, Y., GUO, Z., TIAN, K., and SHEN, X., "On-the-fly elimination of dynamic irregularities for gpu computing," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ACM, 2011.

[105] ZHANG, Y. and OWENS, J. D., "A quantitative performance analysis model for gpu architectures," in *17th International Conference on High-Performance Computer Architecture (HPCA-17)*, (San Antonio, TX, USA), pp. 382–393, IEEE Computer Society, February 2011.

VITA

Naila Farooqui was born to Dr. Mohammad Yahya Farooqui and Dr. Zakia Jabbar Farooqui in 1982. She was brought up in Riyadh, Saudi Arabia and attended the Saudi Arabian International School-Riyadh (SAIS-R). She received her Bachelor's Degree in Computer Science from the Georgia Institute of Technology in 2003. After working in the industry as a software engineer, Naila returned to Georgia Tech and completed her Ph.D. in Computer Science, under the supervision of Dr. Karsten Schwan and Dr. Sudhakar Yalamanchili, in 2015.

Naila's current research focuses on improving the performance and programmability of high-performance computing systems, leveraging changes in both computer architectures and programming models. She has cross-disciplinary expertise in parallel and high-performance computer architectures, compilers and runtime systems, and programming languages.